AFRL-IF-RS-TR-2003-148
Final Technical Report
June 2003

DYNAMITE PROJECT AUTONOMOUS NEGOTIATING TEAMS (ANTS) PROGRAM

University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J125

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-148 has been reviewed and is approved for publication.

APPROVED: Daniel Daskiewich
DANIEL E. DASKIEWICH
Project Engineer

FOR THE DIRECTOR: Michael Talbert
MICHAEL L. TALBERT, Maj., USAF
Technical Advisor, Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** JUNE 2003	**3. REPORT TYPE AND DATES COVERED** Final May 99 – Mar 03

4. TITLE AND SUBTITLE
DYNAMITE PROJECT AUTONOMOUS NEGOTIATING TEAMS (ANTS) PROGRAM

5. FUNDING NUMBERS
C - F30602-99-2-0507
PE - 62301E
PR - H353
TA - 01
WU - 01

6. AUTHOR(S)
Milind Tambe, Wei-Min Shen, Pagnesh Jay Modi, and Hyuckchul Jung

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Information Science Institute
4676 Admiralty Way
Marina del Rey California 90292

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington Virginia 22203-1714

AFRL/IFTB
525 Brooks Road
Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-148

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Daniel E. Daskiewich/IFTB/(315) 330-7731/ Daniel.Daskiewich@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Distributed resource allocation is a general problem in which a set of agents must intelligently assign their resources to a set of dynamic tasks. Obtaining effective performance in dynamic domains is difficult because agents must deal with significant challenges such as coping with limited resources, ability to respond to changes in tasks, coordinating with other agents and adhering to domain-imposed communication restraints. As part of the DARPA Autonomous Negotiating Teams (ANTs) program, this project addressed shortcomings in current distributed resource allocation research. Significant advancements were made in three areas: 1) Creation of abstract formalizations of the problem that allows the development of general solution strategies, 2) Development of the Adopt algorithm, which addresses solving dynamic distributed constraint problems, and 3) Application of the techniques in a practical domain involving real-world hardware. Target tracking in distributed sensor networks is the selected application area addressed by this report.

14. SUBJECT TERMS
Distributed Resource Allocation, Distributed Constraint Satisfaction Problems, Intelligent Software Agents

15. NUMBER OF PAGES
52

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Figures

1 Overview

Distributed resource allocation is a general problem in which a set of agents must intelligently assign their resources to a set of dynamic tasks. Target tracking in distributed sensor networks is one example of a real-world domain where this problem arises. Other examples include distributed scheduling [3], supply chains [20], unmanned UAVs [1] and urban disaster rescue [9]. Obtaining effective performance in these domains is difficult because agents must deal with significant challenges such as coping with limited resources, ability to respond to changes in tasks, coordinating with other agents and adhering to domain-imposed communication constraints.

Despite the significant progress in distributed resource allocation, there are currently three key shortcomings in this research. First, there is a lack of abstract formalizations of the problem that allow development of general solution strategies. In particular, we desire formalizations that allow the general mapping of different problem types into well-known problem solving paradigms, in our case Distributed Constraint Reasoning (DCR) [21][23][12][18]. DCR has been shown to be an effective general way to model and solve complex distributed problems [14] [8] [2] [19] [16] [15]. Unfortunately, general guidance on how to represent a given distributed resource allocation problem within the DCR paradigm is currently missing.

Second, existing methods for DCR are insufficient for addressing the challenges present in the distributed resource allocation problem. In particular, previous work in DCR has dealt mainly with Distributed Constraint Satisfaction Problems (DisCSP) [21]. While this is an important advancement, DisCSP is not adequate for representing problems where solutions may have degrees of quality or cost. For example, coping with limited resources requires techniques for solving overconstrained problems. In DisCSP, overconstrained problems are simply designated as unsolvable. Instead, we require representations and algorithms for DCR that allow agents to reason about solutions with degrees of quality.

The third problem is whether and how DCR algorithms developed on abstract problems can be applied in practical domains involving real-world hardware. In current research, researchers often abstract out key real-world coordination problems for systematic investigation within software testbeds[5], unhindered by other complex factors in real-world environments. While such abstraction is necessary, it is also important to understand the techniques and principles to apply multiagent algorithms to real-world domains, where uncertainty, real-time constraints and dynamism prevail. Identifying the principles for applying multiagent algorithms to real-world domains will help in developing real applications and may assist in identifying key weaknesses of existing multiagent algorithms.

We report on advances in addressing the above three issues in distributed resource allocation. Figure 1 depicts the overall methodology. First, to address the lack of formalizations and general solution strategies, we propose a formalization of distributed resource allocation that is expressive enough to represent two of its salient features, distributedness and dynamism. In the formalization, agents execute operations to detect and perform tasks that arise over time in the environment. The agents must correctly allocate themselves to tasks so that all tasks are performed. We present two reusable, generalized mappings that automatically convert a given distributed resource allocation problem into a DCR representation. Each mapping is proven to correctly represent resource allocation problems of specific difficulty.

Next, to address the problem of reasoning in overconstrained situations, we propose a new algorithm, called *Adopt*, for Distributed Constraint Optimization Problems (DCOP). DCOP models a global objective function as a set of *valued* constraints. In other words, constraints are described as functions that return a range of values, rather than predicates that return only true or false. DCOP significantly generalizes the DisCSP framework mentioned above. Adopt, to the best of our knowledge, is the first algorithm for DCOP that can find either an optimal solution or a solution within a user-specified distance from the optimal, using only localized asynchronous communication and polynomial space at each agent. The algorithm is shown to significantly outperform its competitors in terms of efficiency in finding solutions on benchmark problems.

Finally, to address the problem of applying DCR algorithms to real-world hardware, we report on what was necessary to apply Adopt to the distributed sensor network problem. In particular, we developed a two-layered architecture. The general principle is to allow the DCR coordination algorithm to work as a higher layer coordinating inter-agent activities while the lower layer is a probabilistic component that deals with task uncertainty and dynamics and allows an agent to do local reasoning when time for coordination is not available. We present results obtained from an implementation of the system described on hardware sensors. The results show that the two-layered architecture is effective at tracking moving targets and the probability model and local reasoning enabled the multiagent algorithm to deal with the difficulties posed by the distributed sensor domain. We believe these results are the first successful application of DCR on real hardware. We believe this is a significant first step towards moving multiagent algorithms developed on abstract problems onto real hardware.

2 Application Domain

We describe the distributed sensor network problem which is used to both illustrate and validate our approach. The domain consists of multiple stationary sensors, each controlled by an independent agent, and targets moving through their sensing range. Figure 2.a shows the hardware and simulator screen, respectively. Each sensor is equipped with a Doppler radar with three sector heads. Each sector head covers 120 degrees and only one sector can be active at a time. While all of the sensor agents must choose to activate their sector heads to track the targets, there are some key difficulties in such tracking.

The first difficulty is that the domain is inherently distributed. In order for a target to be tracked accurately, at least three agents must collaborate. They must concurrently activate their sectors so the target is sensed by at least three overlapping sectors. For example, in Figure 2.b which corresponds to the simulator in Figure 2.a, if an agent A1 detects target 1 in its sector 0, it must inform two of its neighboring agents, A2 and A4 for example, so that they activate their respective sectors that overlap with A1's sector 0.

The second difficulty with accurate tracking is that when an agent is informed about a target, it may face ambiguity in which sector to activate. Each sensor can detect only the distance and speed of a target, so an agent that detects a target can only inform another agent about the general area of where a target may be, but cannot tell other agents specifically which sector they must activate. For example, suppose there is only target 1 in Figure 2.b and agent A1 detects that a target is present in its sector 0. A1 can tell A2 that a target is somewhere in the region of its sector 0, but it cannot tell A2 which sector to activate because A2 has two sectors (sector 1 and 2) that overlap with A1's

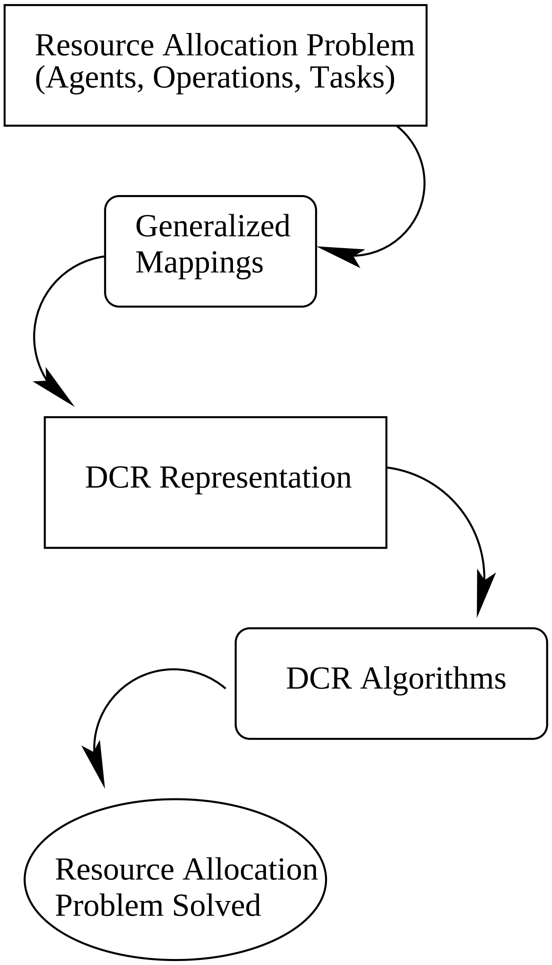
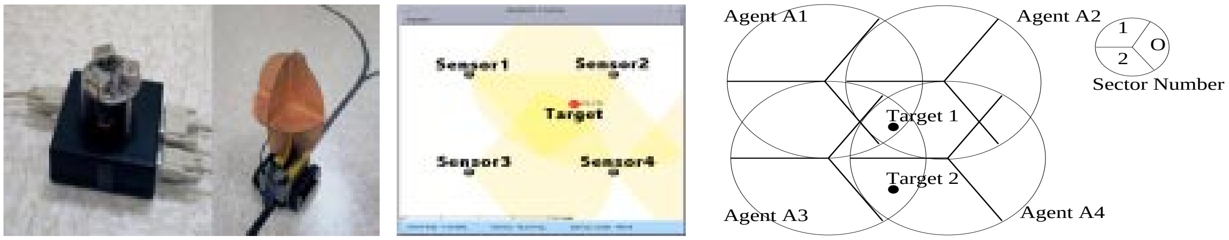


Figure 1: Graphical depiction of the described methodology.



(a) hardware and simulator

(b) sensor sectors

Figure 2: A distributed sensor network for tracking moving targets.

sector 0. In order to resolve this ambiguity, A2 may be forced to first activate its sector 1, detect no target, then try its sector 2. Thus, activating the correct sector requires a collaboration between A1 and A2. A1 informs A2 that a target exists in some ambiguous region and A2 then resolves the remaining ambiguity. The significance here is that no single agent can determine the correct allocation of all sectors to targets.

The third difficulty is that resource contention may occur when multiple targets must be tracked simultaneously. For instance, in Figure 2.b, A4 needs to decide whether to track target 1 or target 2 and it cannot do both since it may activate only one sector at a time. A4 should choose to track target 2 since there is no other way for target 2 to be tracked. A4 is “critical” for tracking target 2. In general, determining whether an agent is “critical” for a particular target requires non-local information about targets out of an agent’s immediate sensing range. In this example, note that target 1 and 2 are tasks that conflict with one another. Targets that are spatially distant do not conflict with each other and thus can easily be tracked without resource contention. Thus, as we will see, the relationship among tasks will affect the difficulty of the overall resource allocation problem.

Finally, the situation is dynamic because targets move through the sensing range. Even after agents find a configuration that is accurately tracking all targets, they may have to reconfigure themselves as targets move over time.

While we will focus mainly on the distributed sensor network problem, a second domain which motivates our work is large-scale urban disaster recovery, e.g., performing search and rescue operations after an earthquake. Robocup Rescue [9] is a detailed simulation environment of the 1995 Kobe, Japan earthquake in which over 5000 people were killed. In this simulator, multiple fire engines, ambulances and police cars must collaborate to save trapped civilians from burning buildings. Centralized control may not be available to allocate all of the emergency response resources since communication infrastructure may be damaged or overloaded. Individual agents must communicate locally and collaborate with one another in order to allocate their resources correctly. For instance, an ambulance agent which must rescue a civilian trapped in a burning building must collaborate with a fire engine agent who is able to extinguish the fire. The tasks are dynamic, e.g., fires grow or shrink and also ambiguous e.g., a fire engine could receive a report of a fire in an area, but not a specific building on fire. This domain thus presents another example of a distributed resource allocation problem with many similarities to the distributed sensor network problem.

The above applications illustrate the difficulty of resource allocation among distributed agents in a dynamic environment. Lack of a formalism for dynamic distributed resource allocation problem can lead to ad-hoc methods which cannot be easily reused. Instead, our adoption of a formal model allows our problem and its solution to be stated in a more general way, possibly increasing our solution’s usefulness. More importantly, a formal treatment of the problem also allows us to study its complexity and provide other researchers with some insights into the difficulty of their own resource allocation problems. Finally, a formal model allows us to provide guarantees of soundness and completeness of our results. The next section presents our formal model of resource allocation.

3 Modeling Multiagent Resource Allocation via Distributed Constraint Satisfaction

Multiagent resource allocation problems are difficult because they are both *distributed* and *dynamic*. First, a key implication of the distributed nature of this problem is that the control is distributed in multiple agents; yet these multiple agents must collaborate to accomplish the tasks at hand. Second, another implication is that agents face *global ambiguity*. An agent may know, based on the results of its local operations, that some task (out of a possible set of tasks) is present. However, it may not be able to individually determine exactly which task is present. The agents must collaborate to determine which one of the many possible tasks is actually present and needs to be done. Third, different tasks may require the same resources and thus, *resource contention* may occur. In these situations, agents must take care to allocate critical resources to appropriate tasks. Allocating a critical resource incorrectly may lead to situations where some other tasks must go unperformed. Finally, the situation is dynamic so a particular allocation of resources at one time may become obsolete when the underlying tasks have changed. The agents must have a way to express and cope with such changes in the problem.

In this section, we attempt to develop a systematic formalization of the problem and general solution strategies for the types of problems modeled. First, we propose a formalization of distributed resource allocation that is expressive enough to represent some salient features of the problem. This formalization is significant because it allows us to understand the complexity of different types of resource allocation problems and may also enable future researchers to understand the difficulty of their own resource allocation problem. Second, in order to solve these types of resource allocation problems, we define the notion of a Dynamic Distributed Constraint Satisfaction Problem (DyDisCSP). While this section does not focus on new algorithms for DisCSP, it does focus on applying DisCSP in service of distributed resource allocation problems. In dynamic domains where features of the environment may not be known in advance, it is difficult to completely specify a DisCSP problem in advance. To address this difficulty, DyDisCSP generalizes DisCSP by allowing agents to add or remove local constraints from the problem as external environmental conditions change.

Given the distributed resource allocation formalism and the DyDisCSP problem, we present two reusable, generalized mappings, Mapping I and II, that automatically convert a given distributed resource allocation problem into a DyDisCSP. Each mapping is proven to correctly perform resource allocation problems of specific difficulty. These generalized mappings enable existing distributed constraint reasoning technologies to be brought to bear directly onto the distributed resource allocation problem. In addition, they allow future advances in distributed constraint reasoning to also be directly applied to the distributed resource allocation problem without significant re-modeling effort. Thus, our formalism and generalized mappings may provide researchers with tools for both representing and solving their resource allocation problem using distributed constraints reasoning.

3.1 Formal Definitions

A Distributed Resource Allocation Problem consists of 1) a set of agents that can each perform some set of operations, and 2) a set of tasks to be completed. In order to be completed, a task requires some subset of agents to perform certain operations. We can define a task by the operations that agents must perform in order to complete it. The problem to be solved is an allocation of agents to tasks such that all tasks are performed. This problem is formalized next.

- **Definition 1**: A Distributed Resource Allocation Problem is a structure $<\mathcal{A}g$, Ω, $\Theta>$ where
  - $\mathcal{A}g$ = $\{A_1$, A_2, ..., $A_n\}$ is a set of agents.
  - Ω = $\{O_1^1, O_2^1$, ..., O_p^i, ..., $O_q^n\}$ is a set of operations, where operation O_p^i denotes the p‘th operation of agent A_i. An operation can either succeed or fail. Let $Op(A_i)$ denote the set of operations of A_i. Operations in $Op(A_i)$ are *mutually exclusive*; an agent can only perform one operation at a time.
  - $\Theta = \{T_1, T_2, ..., T_n\}$ is a set of tasks, where each task $T \in \Theta$ is a set of non-empty sets $\{t_1, t_2, ..., t_n\}$ and each $t_i \in T$ is a set of operations. Each task must satisfy the following property: $\forall t_r$, $t_s \in T$, $t_r \not\subseteq t_s$ and $t_s \not\subseteq t_r$. This requires that each set of operations in a task should be minimal in the sense that no other set is a subset of it. Thus, each t_i is called a *minimal set*. Two minimal sets *conflict* if they contain an operation belonging to the same agent.

Intuitively, the minimal sets of a task specify the alternative ways (sets of operations) to complete the task. A *solution* to a resource allocation problem then, involves choosing a minimal set for each present task such that the minimal sets do not conflict. In this way, when agents perform the operations in those minimal sets, all present tasks are successfully completed. One key difficulty lies in assigning the correct agents to each task so that all tasks get their required resources. An incorrect allocation of resources to one task may result in a shortage of resources for some other task.

To illustrate this formalism in the distributed sensor network domain, we cast each sensor as an agent and activating one of its (three) sectors as an operation. We will use O_p^i to denote the operation of agent A_i activating sector p. For example, in Figure 2.b, we have four agents, so $\mathcal{A}g$ = $\{A_1$, A_2, A_3, $A_4\}$. Each agent can perform one of three operations, so Ω = $\{O_0^1$,O_1^1, O_2^1,O_0^2,O_1^2, O_2^2, O_0^3,O_1^3, O_2^3, O_0^4,O_1^4, O_2^4 $\}$. To specify the subset of operations belonging to a particular agent, say A_1, we use $Op(A_1)$ = $\{$ O_0^1,O_1^1, O_2^1 $\}$.

We now define our task set Θ. We will define a separate task for each region of overlap of sectors where a target may potentially be present. In this way, the existence of a target in a particular location corresponds to a task that needs to be performed. Regions of overlap that do not currently contain a target are tasks that do not currently need to be performed. In the situation illustrated in Figure 2.b, we have only two targets shown and for simplicity, let us assume these are the only possible target locations. We define our task set Θ = $\{T_1$, $T_2\}$. Remember that each target requires three agents to track it so that its position can be triangulated. Thus, task T_1 requires any three of the four possible agents to activate their correct sector, so we define a minimal set corresponding to all the $\binom{4}{3}$ combinations. Thus, T_1 = $\{\{O_0^1$, O_2^2, $O_0^3\}$, $\{O_2^2$, O_0^3, $O_1^4\}$, $\{O_0^1$, O_0^3, $O_1^4\}$, $\{O_0^1$, O_2^2, $O_1^4\}\}$. Note that the subscript of the operation denotes the number of the sector the agent must activate. In the example, task T_2 can only be tracked by two agents. For simplicity, let us assume this is sufficient and T_2 = $\{\{O_0^3$, $O_2^4\}\}$.

For each task, we use $\Upsilon(T_r)$ to denote the union over all the minimal sets of T_r, and for each operation, we use $T(O_p^i)$ to denote the set of tasks T_r that may require O_p^i, i.e., include O_p^i in $\Upsilon(T_r)$. For instance, $\Upsilon(T_1)$ = $\{O_0^1, O_2^2, O_0^3, O_1^4\}$ and $T(O_0^3)$ = $\{ T_1, T_2\}$ in the example above.

As mentioned above, all the tasks in Θ may not always be present. We use $\Theta_{current}$ ($\subseteq \Theta$) to denote the set of tasks that are currently present and require resources. This set is determined by the environment. We call a resource allocation problem *static* if $\Theta_{current}$ is constant over time and *dynamic* otherwise. In our distributed sensor network example, since the targets move and tasks change, the problem is a dynamic one. We do not model resource allocation problems where the resources may be dynamic (i.e., where agents may come and go). Agents must determine which tasks are currently present by executing their operations. The success of an operation is determined by the set of tasks that are currently present in the environment. The following definition formalizes this interface with the environment.

- **Definition 2**: $\forall$ $O_p^i \in \Omega$, if O_p^i is executed and $\exists$ $T_r \in \Theta_{current}$ such that $O_p^i \in \Upsilon(T_r)$, then O_p^i is said to *succeed*. If an operation is executed, but it has no corresponding task in $\Theta_{current}$, the operation is said to *fail*.

In our example, if agent A_1 executes operation O_0^1 (activates sector 1) and if $T_1 \in \Theta_{current}$ (target 1 is present), then O_0^1 will succeed (A_1 will detect a target), otherwise it will fail. In other domains such as disaster rescue, ambulance agents may use sensors that detect the presence of life in a building in order to detect civilians that need medical attention (tasks). Note that our notion of operation failure corresponds to a sensor signal indicating the absence of a task, not actual hardware failure. Hardware failure or sensor noise is an issue not modeled in our formalism. However, an actual system built using this formalism, described later, has been able to incorporate techniques for dealing with noise and failure by using a two-layered architecture, whereby a lower layer of the implementation deals with these issues[17].

A dynamic problem requires agents to detect new tasks as they appear in the environment. This can be done in the distributed sensor domain by agents “scanning” for targets by activating different sectors when they are currently not tracking any target. In disaster rescue, ambulances can drive around searching for injured civilians. This notification procedure is outside of our formalism and we rely on the following assumption.

- **Notification assumption**:
  - i) $\forall T_r \in \Theta$, if $T_r \in \Theta_{current}$, then $\exists$ $O_p^i \in \Upsilon(T_r)$ such that O_p^i succeeds.
  - ii) $\forall T_s (\neq T_r) \in \Theta_{current}$, $O_p^i \notin \Upsilon(T_s)$.

(i) states that if a task is present, at least one agent is notified of the task by the success of one of its operations, i.e., no present task goes unnoticed by everyone. (ii) states that the notifying operation O_p^i (from (i)) must not be required for any other present task. This implies that the success of operation O_p^i will uniquely identify the task T_r among all present tasks. This assumption prevents the possibility whereby the success of a single operation serves to notify an agent of two present tasks. For example, in distributed sensor networks, hardware restrictions preclude this possiblity. Two targets simultaneously present in a single sector results in a garbled signal received by the sensor. Note that the ambiguity problem (figuring out which tasks are present and not present) is not eliminated because we do *not* require the task to be uniquely identified among all (e.g., not present) tasks. Similarly, other difficulties such as resource contention, are not eliminated by this assumption.

Finally, a task is *performed* (the target is tracked) when all the operations in some minimal set succeed (enough radar sectors are sensing the target). More formally,

- **Definition 3**: $\forall T_r \in \Theta$, T_r is *performed* iff there exists a minimal set $t_r \in T_r$ such that all the operations in t_r succeed. A task that is not present cannot be performed, or equivalently, a task that is performed must be included in $\Theta_{current}$.

For example, task T_2 is performed if and only if A_3 executes operation O_0^3 and $A4$ executes operation O_2^4 and both operations succeed simultaneously.

3.2 Properties of Resource Allocation

We now state some definitions that will allow us to categorize a given resource allocation problem and analyze its difficulty. In particular, we notice some properties of task and inter-task relationships. Definitions 4 through 7 are used to describe the complexity of a given task in a given problem, i.e., the definitions relate to properties of an individual task. Next, definitions 8 through 10 are used to describe the complexity of inter-task relationships, i.e., the definitions relate to the interactions between a set of tasks.

3.2.1 Task Complexity

One class of resource allocation problems have the property that each task requires any k agents from a pool of n ($n \geq k$) available agents. That is, the task contains a minimal set for each of the $\binom{n}{k}$ combinations. The following definition formalizes this notion.

- **Definition 4**: $\forall\ T_r \in \Theta$, T_r is **task-$\binom{n}{k}$-exact** iff T_r has exactly $\binom{n}{k_r}$ minimal sets of size k_r, where n = $\mid \Upsilon(T_r) \mid$ and $k_r (\leq n)$ depends on T_r.

For example, the task T_1 (corresponding to target 1 in Figure 2.b) is task-$\binom{4}{3}$-exact because it has exactly $\binom{4}{3}$ minimal sets of size $k = 3$, where $n = 4 = \mid \Upsilon(T_1) \mid$. The following definition defines the class of resource allocation problems where every task is task-$\binom{n}{k}$-exact.

- **Definition 5** : $\binom{n}{k}$**-exact** denotes the class of resource allocation problems $<\mathcal{A}g$, Ω, $\Theta >$ such that $\forall\ T_r \in \Theta$, T_r is task-$\binom{n}{k_r}$-exact.

We find it useful to define a special case of $\binom{n}{k}$-exact resource allocation problems, namely those when $k = n$. Intuitively, all agents are required so each task contains only a single minimal set.

- **Definition 6**: $\binom{n}{n}$**-exact** denotes the class of resource allocation problems $<\mathcal{A}g$, Ω, $\Theta>$ such that $\forall\ T_r \in \Theta$, T_r is task-$\binom{n_r}{k_r}$-exact, where $n_r = k_r = \mid \Upsilon(T_r) \mid$.

For example, the task T_2 (corresponding to target 2 in Figure 2.b) is task-$\binom{2}{2}$-exact.

- **Definition 7**: **Unrestricted** denotes the class of resource allocation problems $<\mathcal{A}g$, Ω, $\Theta>$ with no restrictions on tasks.

Note that $\binom{n}{n}$-exact $\subset$ $\binom{n}{k}$-exact $\subset$ Unrestricted.

3.2.2 Task Relationship Complexity

The following definitions refer to relations between tasks. We define two types of *conflict-free* to denote resource allocation problems that have solutions, or equivalently, problems where all tasks can be performed concurrently.

- **Definition 8**: A resource allocation problem is called **Strongly Conflict Free (SCF)** if for all $T_r, T_s \in \Theta_{current}$ and $\forall\ A_i \in \mathcal{A}g$, $\mid Op(A_i) \cap \Upsilon(T_r) \mid + \mid Op(A_i) \cap \Upsilon(T_s) \mid \leq 1$, i.e., no two tasks have in common an operation from the same agent.

The SCF condition implies that we can choose any minimal set out of the given alternatives for a task and be guaranteed that it will lead to a solution where all tasks are performed, i.e., no backtracking is ever required to find a solution.

- **Definition 9**: A resource allocation problem is called **Weakly Conflict Free (WCF)** if there exists some choice of minimal set for every present task such that all the chosen minimal sets are non-conflicting.

The WCF condition is much weaker that the SCF condition since it only requires that there exists some solution. However, a significant amount of search may be required to find it. Finally, we define problems that may not have a solution.

- **Definition 10**: A resource allocation problem that cannot assumed to be WCF is called (possibly) **over-constrained (OC)**. In OC problems, all tasks may not necessarily be able to be performed concurrently because resources are insufficient.

Note that SCF $\subset$ WCF $\subset$ OC.

3.3 Subclasses of Resource Allocation

Given the above properties, we can define 9 subclasses of problems according to their task complexity and inter-task relationship complexity: SCF and $\binom{n}{n}$-exact, SCF and $\binom{n}{k}$-exact, SCF and unrestricted, WCF and $\binom{n}{n}$-exact, WCF and $\binom{n}{k}$-exact, WCF and unrestricted, OC and $\binom{n}{n}$-exact, OC and $\binom{n}{k}$-exact, OC and unrestricted.

Table 1 summarizes our complexity results for the subclasses of resource allocation problems just defined. The columns of the table, from top to bottom, represent increasingly complex tasks. The rows of the table, from left to right, represent increasingly complex inter-task relationships. Detailed proofs are presented in [14].

Although our formalism and mappings addresses dynamic problems, our complexity analysis here deals with a static problem. A dynamic resource allocation problem can be cast as solving a sequence of static problems, so a dynamic problem is at least as hard as a static one. Furthermore, since distributed problem solving is no easier than a centralized approach (due to communication delays, limited communication range/bandwith, etc.), all our complexity results are based on a centralized problem solver.

Table 1: Complexity Classes of Resource Allocation, n = size of task set Θ, m = size of operation set Ω. Columns represent task complexity and rows represent inter-task relationship complexity.

	SCF	WCF	OC
$\binom{n}{n}$-exact	O(n)	O(n)	NP-Complete
$\binom{n}{k}$-exact	O(n)	O($(n+m)^3$)	NP-Complete
unrestricted	O(n)	NP-Complete	NP-Complete

3.4 Dynamic Distributed CSP

In order to solve general resource allocation problems that conform to our formalized model, we will use distributed constraint satisfaction techniques. Existing approaches to distributed constraint satisfaction fall short for our purposes because they cannot capture the dynamic aspects of the problem. In dynamic problems, a solution to the resource allocation problem at one time may become obsolete when the underlying tasks have changed. This means that once a solution is obtained, the agents must continuously monitor it for changes and must have a way to express such changes in the problem. In order to address this shortcoming, the following section defines the notion of a Dynamic Distributed Constraint Satisfaction Problem (DyDisCSP).

A Constraint Satisfaction Problem (CSP) is commonly defined by a set of variables, each associated with a finite domain, and a set of constraints on the values of the variables. A solution is the value assignment for the variables which satisfies all the constraints. A distributed CSP is a CSP in which variables and constraints are distributed among multiple agents. Each variable belongs to an agent. A constraint defined only on variables belonging to a single agent is called a *local constraint*. In contrast, an *external constraint* involves variables of different agents. Solving a DisCSP requires that agents not only solve their local constraints, but also communicate with other agents to satisfy external constraints.

DisCSP assumes that the set of constraints are fixed in advance. This assumption is problematic when we attempt to apply DisCSP to domains where features of the environment are not known in advance and must be sensed at run-time. For example, in distributed sensor networks, agents do not know where the targets will appear. This makes it difficult to specify the DisCSP constraints in advance. Rather, we desire agents to sense the environment and then activate or deactivate constraints depending on the result of the sensing action. We formalize this idea next.

We take the definition of DisCSP one step further by defining Dynamic DCSP (DyDisCSP). A DyDisCSP is a DisCSP where constraints are allowed to be dynamic, i.e., agents are able to add or remove constraints from the problem according to changes in the environment. More formally,

- **Definition 11**: A *dynamic* constraint is given by a tuple (P, C), where P is an arbitrary predicate that is evaluated by an agent sensing its environment and C is a familiar constraint in DisCSP.

When P is true, C must be satisfied in any DyDisCSP solution. When P is false, C may be violated. An important consequence of dynamic DisCSP is that agents no longer terminate when they reach a stable state. They must continue to monitor P, waiting to see if it changes. If its value changes, they may be required to search for a new solution. Note that a solution when P is true is also a solution when P is false, so the deletion of a constraint does not require any extra computation. However, the converse does not hold. When a constraint is added to the problem, agents may be forced to compute a new solution. In this work, we only need to address a restricted form of DyDisCSP i.e. it is only necessary that *local constraints* be dynamic.

AWC [22] is a sound and complete algorithm for solving DisCSPs. An agent with local variable A_i, chooses a value v_i for A_i and sends this value to agents with whom it has external constraints. It then waits for and responds to messages. When the agent receives a variable value (A_j = v_j) from another agent, this value is stored in an AgentView. Therefore, an AgentView is a set of pairs $\{(A_j, v_j)$, $(A_k,\ v_k)$, ...$\}$. Intuitively, the AgentView stores the current value of non-local variables. A subset of an AgentView is a "NoGood" if an agent cannot find a value for its local variable that satisfies all constraints. For example, an agent with variable A_i may find that the set $\{(A_j, v_j)$, $(A_k,\ v_k)\}$ is a NoGood because, given these values for A_j and A_k, it cannot find a value for A_i that satisfies all of its constraints. This means that these value assignments cannot be part of any solution. In this case, the agent will request that the others change their variable value and a search for a solution continues. To guarantee completeness, a discovered NoGood is stored so that that assignment is not considered in the future.

The most straightforward way to attempt to deal with dynamism in DisCSP is to consider AWC as a subroutine that is invoked anew everytime a constraint is added. Unfortunately, in domains such as ours, where the problem is dynamic but does not change drastically, starting from scratch may be prohibitively inefficient. Another option, and the one that we adopt, is for agents to continue their computation even as local constraints change asynchronously. The potential problem with this approach is that when constraints are removed, a stored NoGood may now become part of a solution. We solve this problem by requiring agents to store their own variable values as part of non-empty NoGoods. For example, if an agent with variable A_i finds that a value v_i does not satisfy all constraints given the AgentView $\{(A_j, v_j)$, $(A_k,\ v_k)\}$, it will store the set $\{(A_i, v_i)$, (A_j, v_j), $(A_k,\ v_k)\}$ as a NoGood. With this modification to AWC, NoGoods remain "no good" even as local constraints change. Let us call this modified algorithm Locally-Dynamic AWC (LD-AWC) and the modified NoGoods "LD-NoGoods" in order to distinguish them from the original AWC NoGoods. The following lemma establishes the soundness and completeness of LD-AWC.

Lemma I: LD-AWC is sound and complete.

The soundness of LD-AWC follows from the soundness of AWC. The completeness of AWC is guaranteed by the recording of NoGoods. A NoGood logically represents a set of assignments that leads to a contradiction. We need to show that this invariant is maintained in LD-NoGoods. An LD-NoGood is a superset of some non-empty AWC NoGood and since every superset of an AWC NoGood is no good, the invariant is true when a LD-NoGood is first recorded. The only problem that remains is the possibility that an LD-NoGood may later become good due to the dynamism of local constraints. A LD-NoGood contains a specific value of the local variable that is no good but never contains a local variable exclusively. Therefore, it logically holds information about external constraints only. Since external constraints are not allowed to be dynamic in LD-AWC, LD-NoGoods remain valid even in the face of dynamic local constraints. Thus the completeness of LD-AWC is guaranteed.

3.5 Mapping SCF Problems into DyDisCSP

We now describe a solution to the SCF subclass of resource allocation problems, defined in Definition 8 of Section 3.1, by mapping onto DyDisCSP. Our goal is to provide a general mapping, named Mapping I, that allows any unrestricted SCF resource allocation problem to be modeled as DyDisCSP by applying this mapping.

Mapping I is motivated by the following idea. The goal in DyDisCSP is for agents to choose values for their variables so all constraints are satisfied. Similarly, the goal in resource allocation is for the agents to choose operations so all tasks are performed. Therefore, in our first attempt we map variables to agents and values of variables to operations of agents. For example, if an agent A_i has three operations it can perform, $\{O^i_1, O^i_2, O^i_3\}$, then the variable corresponding to this agent will have three values in its domain. However, this simple mapping attempt fails due to the dynamic nature of the problem; operations of an agent may not always succeed. Therefore, we define two values for every operation, one for success and the other for failure. In our example, this would result in six values for each variable A_i: $\{O^i_1$yes, O^i_2yes, O^i_3yes, O^i_1no, O^i_2no, O^i_3no $\}$.

It turns out that even this mapping is inadequate due to ambiguity. Ambiguity arises when an operation can be required for multiple tasks but only one task is actually present. To resolve ambiguity, we desire agents to be able to not only communicate about which operation to perform, but also to communicate for which task they intend the operation. For example in Figure 2.b, Agent A3 is required to activate the same sector for both targets 1 and 2. We want A3 to be able to distinguish between the two targets when it communicates with A2, so that A2 will be able to activate its correct respective sector. For each of the values defined so far, we will define new values corresponding to each task that an operation may serve.

Mapping I: Given a Resource Allocation Problem $\langle \mathcal{A}g$, Ω, $\Theta\rangle$, the corresponding DyDisCSP is defined over a set of n variables.

- A = $\{A_1$, A_2,..., $A_n\}$, one variable for each $A_i \in$ Ag. We will use the notation A_i to interchangeably refer to an agent or its variable.

The domain of each variable is given by:

- $\forall A_i \in \mathcal{A}g$, Dom($A_i$) = $\bigcup_{O^i_p \in \Omega} O^i_pxT(O^i_p)x\{yes,no\}$.

In this way, we have a value for every combination of operations an agent can perform, a task for which this operation is required, and whether the operation succeeds or fails. For example in Figure 2.b, Agent A3 has one operation (sector 0) with two possible tasks (target 1 and 2). Although the figure does not show targets in sector 1 and sector 2 of agent A3, let us assume that targets may appear there for this example. Thus, let task T_3 be defined as a target in A3's sector 1 and let task T_4 be defined as a target in A3's sector 2. This means A3 would have 8 values in its domain: $\{O^3_0T_1$yes, $O^3_0T_1$no, $O^3_0T_2$yes, $O^3_0T_2$no, $O^3_1T_3$yes, $O^3_1T_3$no, $O^3_2T_4$yes, $O^3_2T_4$no $\}$.

A word about notation: $\forall$ $O^i_p \in \Omega$, the set of values in O^i_px$T(O^i_p)$x$\{$yes$\}$ will be abbreviated by the term O^i_p*yes and the assignment A_i = O^i_p*yes denotes that $\exists v \in O^i_p$*yes such that $A_i = v$. Intuitively, the notation is used when an agent detects that an operation is succeeding, but it is not known which task is being performed. This is analogous to the situation in the distributed sensor network domain where an agent may detect a target in a sector, but does not know its exact location. Finally, when a variable A_i is assigned a value, the corresponding agent executes the corresponding operation.

Next, we must constrain agents to assign "yes" values to variables only when an operation has succeeded. However, in dynamic problems, an operation may succeed at some time and fail at another time since tasks are dynamically added and removed from the current set of tasks to be performed. Thus, every variable is constrained by the following *dynamic* local constraints (as defined in Section 3.4).

- **Dynamic Local Constraint 1 (LC1)**: $\forall T_r \in \Theta$, $\forall O_p^i \in \Upsilon(T_r)$,
  LC1(A_i) = (P, C), where Predicate P: O_p^i succeeds.
  Constraint C: $A_i = O_p^i$*yes
- **Dynamic Local Constraint 2 (LC2)**: $\forall T_r \in \Theta$, $\forall O_p^i \in \Upsilon(T_r)$,
  LC2(A_i) = (P, C), where Predicate P: O_p^i does not succeed.
  Constraint C: $A_i \neq O_p^i$*yes

The truth value of P is not known in advance. Agents must execute their operations, and based on the result, locally determine if C needs to be satisfied. In dynamic problems, where the set of current tasks is changing over time, the truth value of P will also change over time, and hence the corresponding DyDisCSP will need to be continually monitored and resolved as necessary.

We now define the External Constraint (EC) between variables of two different agents. EC is a normal static constraint and must always be satisfied.

- **External Constraint**: $\forall T_r \in \Theta$, $\forall O_p^i \in \Upsilon(T_r)$, $\forall A_j \in A$,

  EC(A_i, A_j): (1) A_i = $O_p^i T_r$yes, and
  (2) $\forall t_r \in T_r$, $O_p^i \in t_r$, $\exists q$ $O_q^j \in t_r$.
  $\Rightarrow$ A_j = $O_q^j T_r$yes

The EC constraint requires some explanation. It says that if A_i detects a task, then other agents in minimal set t_r must also help with the task. In particular, Condition (1) states that an agent A_i is executing a successful operation O_p^i for task T_r. Condition (2) quantifies the other agents whose operations are also required for T_r. If A_j is one of those agents, i.e., O_q^j is an operation that can help perform T_r, the consequent requires A_j to choose operation O_q^j. Note that every pair of variables A_i and A_j have an EC constraint between them. If A_j is not required for T_r, condition (2) is false and EC is trivially satisfied.

3.5.1 Correctness of Mapping I

We now show that Mapping I can be used to model a given SCF resource allocation problem as a DyDisCSP. Theorem II states that our DyDisCSP always has a solution. This means the constraints as defined above are not inconsistent and thus, it is always possible to solve the resulting DyDisCSP. Theorem III then states that if agents reach a solution, all tasks are performed. Note that the converse of the Theorem III does not hold, i.e. it is possible for agents to be performing all tasks *before* a solution to the DyDisCSP is reached. This is due to the fact that when all current tasks are being performed, agents whose operations are not necessary for the current tasks could still be violating some constraints.

Theorem II: Given an unrestricted SCF Resource Allocation Problem $\langle \mathcal{A}g,\Omega,\Theta \rangle$, $\Theta_{current} \subseteq \Theta$, a solution always exists for the DyDisCSP obtained from Mapping I.

proof: We proceed by presenting a solution to any given DyDisCSP problem obtained from Mapping I.

Let $B = \{A_i \in A \mid \exists T_r \in \Theta_{current}, \exists O_p^i \in \Upsilon(T_r)\}$. B contains precisely those agents whose have an operation that can contribute to some current task. We will first assign values to variables in B, then assign values to variables that are not in B. If $A_i \in B$, we assign $A_i = O_p^i T_r yes$, where $T_r \in \Theta_{current}$ and $O_p^i \in \Upsilon(T_r)$. We know such T_r and O_p^i exist by the definition of B. If $A_i \notin B$, we may choose any $O_p^i T_r no \in$ Domain(A_i) and assign $A_i = O_p^i T_r no$.

To show that this assignment is a solution, we first show that it satisfies the EC constraint. We arbitrarily choose two variables, A_i and A_j, and show that EC(A_i, A_j) is satisfied. We proceed by cases. Let $A_i, A_j \in A$ be given.

- *case 1:* $A_i \notin B$

  Since $A_i = O_p^i T_r no$, condition (1) of EC constraint is false and thus EC is trivially satisfied.

- *case 2:* $A_i \in B, A_j \notin B$

  $A_i = O_p^i T_r yes$ in our solution. Let $t_r \in T_r$, $O_p^i \in t_r$. We know that $T_r \in \Theta_{current}$ and since $A_j \notin B$, we conclude that $\nexists O_q^j \in t_r$. Condition (2) of the EC constraint is false and thus EC is trivially satisfied.

- *case 3:* $A_i \in B, A_j \in B$

  $A_i = O_p^i T_r yes$ and $A_j = O_q^j T_s yes$ in our solution. Let $t_r \in T_r$, $O_p^i \in t_r$. T_s and T_r must be strongly conflict free since both are in $\Theta_{current}$. If $T_s \neq T_r$, then $\nexists\ O_n^j \in \Omega$, $O_n^j \in t_r$. Condition (2) of EC(A_i,A_j) is false and thus EC is trivially satisfied. If $T_s = T_r$, then EC is satisfied since A_j is helping A_i perform T_r.

Next, we show that our assignment satisfies the LC constraints. If $A_i \in B$ then $A_i = O_p^i T_r yes$, and LC1, regardless of the truth value of P, is clearly not violated. Furthermore, it is the case that O_p^i succeeds, since T_r is present. Then the predicate P of LC2 is not true and thus LC2 is not present. If $A_i \notin B$ and $A_i = O_p^i T_r no$, it is the case that O_p^i is executed and, by definition, does not succeed. Then, the predicate P of LC1 is not satisfied and thus LC1 is not present. LC2, regardless of the truth value of P, is clearly not violated. Thus, the LC constraints are satisfied by all variables. We can conclude that all constraints are satisfied and our value assignment is a solution to the DyDisCSP.

Theorem III: Given an unrestricted SCF Resource Allocation Problem $\langle \mathcal{A}g,\Omega,\Theta \rangle$, $\Theta_{current} \subseteq \Theta$ and the DyDisCSP obtained from Mapping I, if an assignment of values to variables in the DyDisCSP is a solution, then all tasks in $\Theta_{current}$ are performed.

proof: Let a solution to the DyDisCSP be given. We want to show that all tasks in $\Theta_{current}$ are performed. We proceed by choosing a task $T_r \in \Theta_{current}$. Since our choice is arbitrary and tasks are strongly conflict free, if we can show that it is indeed performed, we can conclude that all members of $\Theta_{current}$ are performed.

Let $T_r \in \Theta_{current}$ be given. By the **Notification Assumption**, some operation O^i_p, required by T_r will be executed. However, the corresponding agent A_i, will be unsure as to which task it is performing when O^i_p succeeds. This is due to the fact that O^i_p may be required for many different tasks. It may choose a task, $T_s \in T(O^i_p)$, and LC1 requires it to assign the value $O^i_p T_s$yes. We will show that A_i could not have chosen incorrectly since we are in solution state. The EC constraint will then require that all other agents A_j, whose operations are required for T_s also execute those operations and assign $A_j = O^j_q T_s yes$. We are in solution, so LC2 cannot be present for A_j. Thus, O^j_q succeeds. Since all operations required for T_s succeed, T_s is performed. By definition, $T_s \in \Theta_{current}$. But since we already know that T_s and T_r have an operation in common, the Strongly Conflict Free condition requires that $T_s = T_r$. Therefore, T_r is indeed performed.

3.6 Mapping WCF Problems into DyDisCSP

This section begins with a discussion of the difficulty in using Mapping I for solving WCF problems. This leads to the introduction of a second mapping, Mapping II, which is able to map WCF problems into DyDisCSP so that it can be efficiently solved using existing distributed constraint reasoning methods.

Our first mapping has allowed us to solve any SCF resource allocation problem. However, when we attempt to solve WCF resource allocation problems with this mapping, it fails because the DyDisCSP becomes overconstrained. This is due to the fact that Mapping I requires all agents who can possibly help perform a task to do so. If only three out of four agents are required for a task, Mapping I will still require all four agents to perform the task. In some sense, this results in an overallocation of resources to some tasks. This is not a problem when all tasks are independent as in the SCF case. However, in the WCF case, this overallocation may leave other tasks without sufficient resources to be performed. One way to solve this problem is to modify the constraints in the mapping to allow agents to reason about relationships among tasks. However, this requires adding n-ary ($n > 2$) external constraints to the mapping. This is problematic in a distributed situation because there are no efficient algorithms for non-binary distributed CSPs. Existing methods require extraordinary amounts of inter-agent communication. Instead, we create a new mapping by extending mapping I to n-ary constraints, then taking its dual representation. In the dual representation, variables correspond to tasks and values correspond to operations. This allows all n-ary constraints to be *local* within an agent and all external constraints are reduced to equality constraints. Restricting n-ary constraints to be local rather than external is more efficient because it reduces the amount of communication needed between agents. This new mapping, Mapping II, allocates only minimal resources to each task, allowing WCF problems to be solved. Mapping II is described next and proven correct. Here, each agent has a variable for each task in which its operations are included.

Mapping II: Given a Resource Allocation Problem $\langle \mathcal{A}g, \Omega, \Theta \rangle$, the corresponding DyDisCSP is defined as follows:

- **Variables:** $\forall T_r \in \Theta, \forall O^i_p \in \Upsilon(T_r)$, create a DyDisCSP variable $T_{r,i}$ and assign it to agent A_i.

- **Domain:** For each variable $T_{r,i}$, create a value $t_{r,i}$ for each minimal set in T_r, plus a "NP" value (not present). The NP value allows agents to avoid assigning resources to tasks that are not present and thus do not need to be performed.

In this way, we have a variable for each task and a copy of each such variable is assigned to each agent that has operation for that task. For example in Figure 2.b, Agent A1 has one variable, $T_{1,1}$, Agent A2 has one variable $T_{1,2}$, Agent A3 has two variables, $T_{1,3}$ and $T_{2,3}$, one for each task it can perform, and Agent A4 has two variables, $T_{1,4}$ and $T_{2,4}$. The domain of each $T_{1,i}$ variable has five values, one for each of the four minimal sets as described in Section 3.1, plus the NP value.

Next, we must constrain agents to assign non-NP values to variables only when an operation has succeeded, which indicates the presence of the corresponding task. However, in dynamic problems, an operation may succeed at some time and fail at another time since tasks are dynamically added and removed from the current set of tasks to be performed. Thus, every variable is constrained by the following dynamic local constraints.

- **Dynamic Local (Non-Binary) Constraint (LC1)**:

  $\forall A_i \in \mathcal{A}g, \forall O_p^i \in Op(A_i)$, let B = { $T_{r,i} \mid O_p^i \in T_r$ }. Then let the constraint be defined as a non-binary constraint over the variables in B as follows:

  Predicate P: O_p^i succeeds

  Constraint C: $\exists T_{r,i} \in$ B $T_{r,i} \neq$ NP.

- **Dynamic Local Constraint (LC2)**: $\forall T_r \in \Theta, \forall O_p^i \in \Upsilon(T_r)$, let the constraint be defined on $T_{r,i}$ as follows:

  Predicate P: O_p^i does not succeed

  Constraint C: $T_{r,i}$ = NP.

We now define the constraint that defines a valid allocation of resources and the external constraints that require agents to agree on a particular allocation.

- **Static Local Constraint (LC3)**: $\forall T_{r,i}, T_{s,i}$, if $T_{r,i} = t_{r,i}$ and $T_{s,i} = t_{s,i}$, then $t_{r,i}$ and $t_{s,i}$ cannot conflict. NP does not conflict with any value.

- **External Constraint (EC)**: $\forall i, j, r\ T_{r,i} = T_{r,j}$.

For example, if Agent A4 assigns $T_{1,4} = \{O_0^1, O_2^2, O_2^4\}$, then LC3 says it cannot assign a minimal set to its other variable $T_{2,4}$, that contains any operation of either Agent A1, A2 or A4. Since $T_{2,4}$ has only one minimal set, $\{O_0^3, O_2^4\}$ which contains Agent A4, the only compatible value is NP. Note that if Target 1 and 2 are both present simultaneously as shown in Figure 2.b, the situation is overconstrained since the NP value will be prohibited by LC1.

3.6.1 Correctness of Mapping II

We will now prove that Mapping II can be used to represent any given WCF Resource Allocation Problem as a DyDisCSP. As in Mapping I, the Theorem VII shows that our DyDisCSP always has a solution, and the Theorem VIII shows that if agents reach a solution, all current tasks are performed.

Theorem VII: Given a WCF Resource Allocation Problem $\langle\mathcal{A}g,\Omega,\Theta\rangle$, $\Theta_{current} \subseteq \Theta$, there exists a solution to DyDisCSP obtained from Mapping II.

proof: For all variables corresponding to tasks that are not present, we can assign the value "NP". This value satisfies all constraints except possibly LC1. But the P condition must be false since the task is not present, so LC1 cannot be violated. We are guaranteed that there is a choice of non-conflicting minimal sets for the remaining tasks (by the WCF condition). We can assign the values corresponding to these minimal sets to those tasks and be assured that LC3 is satisfied. Since all variable corresponding to a particular task get assigned the same value, the external constraint is satisfied. We have a solution to the DyDisCSP.

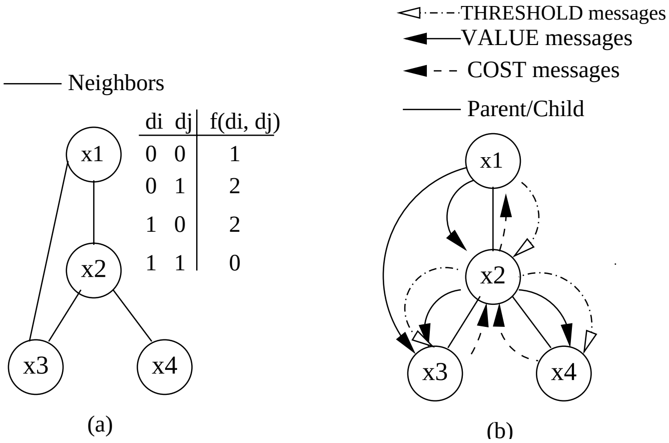
Theorem VIII: Given a WCF Resource Allocation Problem $\langle\mathcal{A}g,\Omega,\Theta\rangle$, $\Theta_{current} \subseteq \Theta$ and the DyDisCSP obtained from Mapping II, if an assignment of values to variables in the DyDisCSP is a solution, then all tasks in $\Theta_{current}$ are performed.

proof: Let a solution to the DyDisCSP be given. We want to show that all tasks in $\Theta_{current}$ are performed. We proceed by contradiction. Let $T_r \in \Theta_{current}$ be a task that is not performed in the given solution state. Condition (i) of the **Notification Assumption** says some operation O^i_p, required by T_r will be executed and (by definition) succeed. LC1 requires the corresponding agent A_i, to assign a minimal set to some task which requires O^i_p. There may be many choices of tasks that require O^i_p. Suppose A_i chooses a task T_s. A_i assigns a minimal set, say t_s, to the variable $T_{s,i}$. The EC constraint will then require that all other agents A_j, who have a local copy of T_s called $T_{s,j}$, to assign $T_{s,j} = t_s$. In addition, if A_j has an operation O^j_q in the minimal set t_s, it will execute that operation. Also, we know that A_j is not already doing some other operation since t_s cannot conflict with any other chosen minimal set (by LC3).

We now have two cases. In case 1, suppose $T_s \neq T_r$. Condition (ii) of the **Notification Assumption** states that T_r is the only task that both requires O^i_p and is actually present. Thus, T_s cannot be present. By definition, if T_s is not present, it cannot be performed. If it cannot be performed, there cannot exist a minimal set of T_s where all operations succeed (def of "performed"). Therefore, some operation in t_s must fail. Let O^j_q be an operation of agent A_j that fails. Since A_j has assigned value $T_{s,j} = t_s$, LC2 is violated by A_j. This contradicts the fact we are in a solution state. Case 1 is not possible. This leaves case 2 where $T_s = T_r$. Then, all operations in t_s succeed and T_r is performed. We assumed T_r was not performed, so by contradiction, all tasks in $\Theta_{current}$ must be performed.

4 Adopt algorithm for DCOP

Existing methods for DCR are insufficient for addressing the challenges present in overconstrained distributed resource allocation problems. In particular, previous work in DCR has dealt mainly with satisfaction based problems, which are inadequate for representing problems where there may be no satisfactory solution. In this section, we present Distributed Constraint Optimization Problems (DCOP) as a generalization of previous DCR representations to allow agents to reason in overconstrained situations. To solve DCOP, we describe a new distributed constraint optimization algorithm, called *Adopt* (Asynchronous Distributed Optimization). Our evaluation results on standard benchmarks show that Adopt obtains several orders of magnitude speed-up over SynchBB, the only existing complete algorithm for DCOP.



di	dj	f(di, dj)
0	0	1
0	1	2
1	0	2
1	1	0

Figure 3: (a) Constraint graph. (b) Communication graph.

DCOP includes a set of variables, each variable is assigned to an agent who has control of its value, and agents must coordinate their choice of values so that a global objective function is optimized. The global objective function is modelled as a set of constraints, and each agent knows about the constraints in which it is involved. We model the global objective function as a set of *valued* constraints, that is, constraints that are described as functions that return a range of values, rather than predicates that return only true or false. Figure 3.a shows an example constraint graph with four agents. In the example, all constraints are identical only for simplicity.

Adopt, to the best of our knowledge, is the first algorithm for distributed constraint optimization that can find either an optimal solution or a solution within a user-specified distance from the optimal, using only localized asynchronous communication and polynomial space at each agent. Detailed proofs are presented in [15]. The main idea behind Adopt is to get asynchrony by allowing each agent to change variable value whenever it detects there is a *possibility* that some other solution may be better than the one currently under investigation. This search strategy allows partial solutions to be abandoned before suboptimality is proved. This increases asynchrony because an agent does not need global information to make its local decisions. The second key idea in Adopt is to efficiently reconstruct previously considered partial solutions (using only polynomial space) through the use of *backtrack threshold*. A backtrack threshold is an allowance on solution cost that prevents backtracking. These two key ideas together yield efficient asynchronous search for optimal solutions. Finally, the third key idea in Adopt is to provide a termination detection mechanism built into the algorithm. The agents terminate whenever they find a complete solution whose cost is under their current backtrack threshold. Previous asynchronous search algorithms have typically required a termination detection algorithm to be invoked separately, which can be problematic since it requires additional message passing.

4.0.2 Overview of Algorithm

Agents are prioritized in a Depth-First Search (DFS) tree in which each agent has a single *parent* and multiple *children*. Thus, unlike previous algorithms such as SynchBB, Adopt does *not* require a linear priority ordering on all the agents. Figure 3.b shows a DFS tree formed from the constraint graph in Figure 3.a. x_1 is the root, x_1 is the parent of x_2 and x_2 is the parent of both x_3 and x_4. Constraints are allowed between an agent and any of its ancestors or descendents (there is a constraint between x_1 and x_3), but there can be no constraints between nodes in different subtrees of the DFS tree. We assume parent and children are neighbors. The requirement of a DFS ordering places no restrictions on the constraint network itself since every connected constraint network can be ordered into some DFS tree[11]. Distributed algorithms for forming DFS trees are also presented in[4] [11]. We will assume the DFS ordering is done in a preprocessing step so every agent knows its parent and children.

The communication in Adopt is shown in Figure 3.b. The algorithm begins by all agents choosing their variable values concurrently. Variable values are sent down constraint edges via VALUE messages. An agent x_i sends VALUE messages only to neighbors lower in the DFS tree and receives VALUE messages only from neighbors higher in the DFS tree. A second type of message, a THRESHOLD message, is sent only from parent to child. A THRESHOLD message contains a single number representing a backtrack threshold, initially zero. Upon receipt of any type of message, an agent i) calculates cost and possibly changes variable value and/or modifies its backtrack threshold, ii) sends VALUE messages to its lower neighbors and THRESHOLD messages to its children and iii) sends a third type of message, a COST message, to its parent. A COST message is sent only from child to parent. A COST message sent from x_i to its parent contains the cost calculated at x_i plus any costs reported to x_i from its children. x_i is informed of costs at agents in the subtree rooted at x_i by COST messages it receives from its own children. To summarize the communication, variable value assignments (VALUE messages) are sent down the DFS tree while cost feedback (COST messages) for the higher agents' value choices percolate back up the DFS tree. It may be useful to view COST messages as a generalization of NoGood message from DisCSP algorithms. THRESHOLD messages are sent down the tree to reduce redundant search.

4.1 Details of Algorithm

Procedures from Adopt are shown in Figure 4 and 5. x_i represents the agent's local variable and d_i represents its current value.

- **Definition:** A *context* is a partial solution of the form $\{(x_j,d_j), (x_k,d_k)...\}$. A variable can appear in a context no more than once. Two contexts are *compatible* if they do not disagree on any variable assignment. $CurrentContext$ is a context which holds x_i's view of the assignments of higher neighbors.

A COST message contains three fields: $context$, lb and ub. The $context$ field of a COST message sent from x_l to its parent x_i contains x_l's $CurrentContext$. This field is necessary because calculated costs are dependent on the values of higher variables, so an agent must attach the context under which costs were calculated to every COST message. This is similar to the *context attachment* mechanism in ABT [21]. When x_i receives a COST message from child x_l, and d is the value of x_i in the $context$ field, then x_i stores lb indexed by d and x_l as $lb(d, x_l)$ (line 32). Similarly, the ub field is stored as $ub(d, x_l)$ and the $context$ field is stored as $context(d, x_l)$ (line 33-34). Before any COST messages are received or whenever contexts become incompatible, i.e., $CurrentContext$ becomes incompatible with $context(d, x_l)$, then $lb(d, x_l)$ is (re)initialized to zero and $ub(d, x_l)$ is (re)initialized to a maximum value Inf (line 3-4, 18-19, 29-30).

x_i calculates cost as local cost plus any cost feedback received from its children. Procedures for calculation of cost are not shown in Figure 4 but are implicitly given by procedure calls, such as **LB** and **UB**, defined next. The *local cost* at x_i, for a particular value choice $d_i \in D_i$, is the sum of costs from constraints between x_i and higher neighbors:

- **Definition:** $\delta(d_i) = \sum_{(x_j,d_j) \in CurrentContext} f_{ij}(d_i, d_j)$ is the *local cost* at x_i, when x_i chooses d_i.

For example, in Figure 3.a, suppose x_3 received messages that x_1 and x_2 currently have assigned the value 0. Then x_3's $CurrentContext$ would be $\{(x_1, 0), (x_2, 0)\}$. If x_3 chooses 0 for itself, it would incur a cost of 1 from $f_{1,3}(0, 0)$ (its constraint with x_1) and a cost of 1 from $f_{2,3}(0, 0)$ (its constraint with x_2). x_3's local cost, $\delta(0) = 1 + 1 = 2$.

When x_i receives a COST message, it adds $lb(d, x_l)$ to its local cost $\delta(d)$ to calculate a *lower bound for value* d, denoted $LB(d)$.

- **Definition:** $\forall d \in D_i, LB(d) = \delta(d) + \sum_{x_l \in Children} lb(d, x_l)$ is a *lower bound* for the subtree rooted at x_i, when x_i chooses d.

Similarly, x_i adds $ub(d, x_l)$ to its local cost $\delta(d)$ to calculate an *upper bound for value* d, denoted $UB(d)$.

- **Definition:** $\forall d \in D_i, UB(d) = \delta(d) + \sum_{x_l \in Children} ub(d, x_l)$ is a *upper bound* for the subtree rooted at x_i, when x_i chooses d.

The *lower bound for variable* x_i, denoted LB, is the minimum lower bound over all value choices for x_i.

- **Definition:** $LB = \min_{d \in D_i} LB(d)$ is a *lower bound* for the subtree rooted at x_i.

Similarly the *upper bound for variable* x_i, denoted UB, is the minimum upper bound over all value choices for x_i.

- **Definition:** $UB = \min_{d \in D_i} UB(d)$ is an *upper bound* for the subtree rooted at x_i.

x_i sends LB and UB to its parent as the lb and ub fields of a COST message (line 52). (Realize that LB need not correspond to x_i's current value, i.e., LB need not equal $LB(d_i)$). Intuitively, $LB = k$ indicates that it is not possible for the sum of the local costs at each agent in the subtree rooted at x_i to be less than k, given that all higher agents have chosen the values in $CurrentContext$. Similarly, $UB = k$ indicates that the optimal cost in the subtree rooted at x_i will be no greater than k, given that all higher agents have chosen the values in $CurrentContext$. Note that if x_i is a leaf agent, it does not receive COST messages, so $\delta(d) = LB(d) = UB(d)$ for all value choices $d \in D_i$ and thus, LB is always equal to UB in every COST message. If x_i is not a leaf but has not yet received any COST messages from its children, UB is equal to maximum value Inf and LB is the minimum local cost $\delta(d)$ over all value choices $d \in D_i$.

x_i's backtrack threshold is stored in the $threshold$ variable, initialized to zero (line 1). Its value is updated in three ways. First, its value can be increased whenever x_i determines that the cost of the optimal solution within its subtree must be greater than the current value of $threshold$. Second, if x_i determines that the cost of the optimal solution within its subtree must necessarily be less than the current value of $threshold$, it decreases $threshold$. These two updates are performed by comparing $threshold$ to LB and UB (lines 53-56, figure 5). The updating of $threshold$ is summarized by the following invariant.

- **ThresholdInvariant:** $LB \leq threshold \leq UB$. The threshold on cost for the subtree rooted at x_i cannot be less than its lower bound or greater than its upper bound.

A parent is also able to set a child's $threshold$ value by sending it a THRESHOLD message. This is the third way in which an agent's $threshold$ value is updated. The reason for this is that in some cases, the parent is able to determine a bound on the optimal cost of a solution within an agent's subtree, but the agent itself may not know this bound. The THRESHOLD message is a way for the parent to inform the agent about this bound.

A parent agent is able to correctly set the $threshold$ value of its children by allocating its own $threshold$ value to its children according to the following two equations. Let $t(d, x_l)$ denote the threshold on cost allocated by parent x_i to child x_l, given x_i chooses value d. Then, the values of $t(d, x_l)$ are subject to the following two invariants.

- **AllocationInvariant:** For current value $d_i \in D_i$, $threshold = \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$. The threshold on cost for x_i must equal the local cost of choosing d plus the sum of the thresholds allocated to x_i's children.

- **ChildThresholdInvariant:** $\forall d \in D_i, \forall x_l \in Children$, $lb(d, x_l) \leq t(d, x_l) \leq ub(d, x_l)$. The threshold allocated to child x_l by parent x_i cannot be less than the lower bound or greater than the upper bound reported by x_l to x_i.

By adhering to these invariants, an agent is able to use its own $threshold$ to determine bounds on the cost of the optimal solution within its childrens' subtrees.

The $threshold$ value is used to determine when to change variable value. Whenever $LB(d_i)$ exceeds $threshold$, x_i changes its variable value to one with smaller lower bound (line 40-41). Such a value necessarily exists since otherwise ThresholdInvariant would be violated. Note that x_i cannot prove that its current value is definitely suboptimal because it is possible that $threshold$ is less than the cost of the optimal solution. However, it changes value to one with smaller cost anyway, thereby realizing the best-first search strategy described previously.

4.2 Example of Algorithm Execution

Figure 6 shows an example of algorithm execution for the DCOP shown in figure 3. Line numbers mentioned in the description refer to figures 4 and 5. This example is meant to illustrate the search process and the exchange of VALUE and COST messages. COST messages are labelled in the figures as [LB,UB,CurrentContext]. For simplicity, not every message sent by every agent is shown. In particular, THRESHOLD messages are omitted from the description. A later example will illustrate how backtrack thresholds are handled.

All agents begin by concurrently choosing a value for their variable (line 5). For this example, let us assume they all choose value 0. Each agent sends its value to all lower priority neighbors (figure 6.a). Since the algorithm is asynchronous, there are many possible execution paths from here. We describe one possible execution path.

x_2 will receive x_1's VALUE message. In line 15, it will record this value into its $CurrentContext$. In line 21, it will enter the **backTrack** procedure. x_2 computes $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 1+0+0 = 1$ and $LB(1) = \delta(1)+lb(1, x_3)+lb(1, x_4) = 2+0+0 = 2$. Since $LB(0) <$

```
initialize
    $threshold \leftarrow 0$; $CurrentContext \leftarrow \{\}$
    forall $d \in D_i, x_l \in Children$ do
        $lb(d, x_l) \leftarrow 0$; $t(d, x_l) \leftarrow 0$
        $ub(d, x_l) \leftarrow Inf$; $context(d, x_l) \leftarrow \{\}$; enddo
    $d_i \leftarrow d$ that minimizes LB($d$)
    backTrack

when received (THRESHOLD, $t$, $context$)
    if $context$ compatible with $CurrentContext$:
        $threshold \leftarrow t$
        maintainThresholdInvariant
        backTrack; endif

when received (TERMINATE, $context$)
    record TERMINATE received from parent
    $CurrentContext \leftarrow context$
    backTrack

when received (VALUE, ($x_j$,$d_j$))
    if TERMINATE not received from parent:
        add ($x_j$,$d_j$) to $CurrentContext$
        forall $d \in D_i, x_l \in Children$ do
            if $context(d, x_l)$ incompatible with $CurrentContext$:
                $lb(d, x_l) \leftarrow 0$; $t(d, x_l) \leftarrow 0$
                $ub(d, x_l) \leftarrow Inf$; $context(d, x_l) \leftarrow \{\}$; endif; enddo
        maintainThresholdInvariant
        backTrack; endif

when received (COST, $x_k$, $context$, $lb$, $ub$)
    $d \leftarrow$ value of $x_i$ in $context$
    remove ($x_i$,$d$) from $context$
    if TERMINATE not received from parent:
        forall ($x_j$,$d_j$) $\in context$ and $x_j$ is not my neighbor do
            add ($x_j$,$d_j$) to $CurrentContext$;enddo
        forall $d' \in D_i, x_l \in Children$ do
            if $context(d', x_l)$ incompatible with $CurrentContext$:
                $lb(d', x_l) \leftarrow 0$; $t(d', x_l) \leftarrow 0$
                $ub(d', x_l) \leftarrow Inf$; $context(d', x_l) \leftarrow \{\}$;endif;enddo;endif
    if $context$ compatible with $CurrentContext$:
        $lb(d, x_k) \leftarrow lb$
        $ub(d, x_k) \leftarrow ub$
        $context(d, x_k) \leftarrow context$
        maintainChildThresholdInvariant
        maintainThresholdInvariant; endif
    backTrack

procedure backTrack
    if $threshold ==$ UB:
        $d_i \leftarrow d$ that minimizes UB($d$)
    else if LB($d_i$) $> threshold$:
        $d_i \leftarrow d$ that minimizes LB($d$);endif
    SEND (VALUE, ($x_i$, $d_i$))
            to each lower priority neighbor
    maintainAllocationInvariant
    if $threshold ==$ UB:
        if TERMINATE received from parent
        or $x_i$ is root:
            SEND (TERMINATE,
            $CurrentContext \cup \{(x_i, d_i)\}$)
                to each child
            Terminate execution; endif;endif
    SEND (COST, $x_i$, $CurrentContext$, LB,UB)
        to parent
```

Figure 4: Procedures for receiving messages (Adopt algorithm). Definitions of terms LB(d), UB(d), LB, and UB are given in the text.

```
procedure maintainThresholdInvariant
if $threshold < \mathbf{LB}$
    $threshold \leftarrow \mathbf{LB}$; endif
if $threshold > \mathbf{UB}$
    $threshold \leftarrow \mathbf{UB}$; endif

%note: procedure assumes ThresholdInvariant is satisfied
procedure maintainAllocationInvariant
while $threshold > \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$ do
    choose $x_l \in Children$ where $ub(d_i, x_l) > t(d_i, x_l)$
    increment $t(d_i, x_l)$; enddo
while $threshold < \delta(d_i) + \sum_{x_l \in Children} t(d_i, x_l)$ do
    choose $x_l \in Children$ where $t(d_i, x_l) > lb(d_i, x_l)$
    decrement $t(d_i, x_l)$; enddo
SEND (THRESHOLD, $t(d_i, x_l)$, $CurrentContext$ )
        to each child $x_l$

procedure maintainChildThresholdInvariant
forall $d \in D_i, x_l \in Children$ do
    while $lb(d, x_l) > t(d, x_l)$ do
        increment $t(d, x_l)$; enddo;endo
forall $d \in D_i, x_l \in Children$ do
    while $t(d, x_l) > ub(d, x_l)$ do
        decrement $t(d, x_l)$; enddo;enddo
```

Figure 5: Procedures for updating backtrack thresholds

$LB(1)$, we have $LB = LB(0) = 1$. x_2 will also compute $UB(0) = \delta(0) + ub(0, x_3) + ub(0, x_4) = 1 + Inf + Inf = Inf$ and $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 2 + Inf + Inf = Inf$. Thus, $UB = Inf$. In line 38, $threshold$ is compared to UB. Since $threshold = 0$ is not equal $UB = Inf$, the test fails. The test in line 40 succeeds since $LB(0) = 1 > threshold = 0$. In line 41, x_2 will choose a value that minimizes LB, which in this case is still $x_2 = 0$. In line 52, x_2 sends the corresponding COST message to x_1 (figure 6.b).

Concurrently with x_2's execution, x_3 will go through a similar execution. x_3 will evaluate its constraints with higher agents and compute $LB(0) = \delta(0) = f_{1,3}(0,0) + f_{2,3}(0,0) = 1+1 = 2$. A change of value to $x_3 = 1$ would incur a cost of $LB(1) = \delta(1) = f_{1,3}(0,1) + f_{2,3}(0,1) = 2+2 = 4$, so instead x_3 will stick with $x_3 = 0$. x_3 will send a COST message with $LB = UB = 2$, with associated context $\{(x_1, 0), (x_2, 0)\}$, to its parent x_2. x_4 executes similarly (figure 6.b).

Next, x_1 receives x_2's COST message. In line 31, x_1 will test the received context $\{(x_1, 0)\}$ against $CurrentContext$ for compatibility. Since x_1's $CurrentContext$ is empty, the test will pass. (Note that the root never receives VALUE messages, so its $CurrentContext$ is always empty.) The received costs will be stored in lines 32-33 as $lb(0, x_2) = 1$ and $ub(0, x_2) = Inf$. In line 37, execution enters the **backTrack** procedure. x_1 computes $LB(1) = \delta(1) + lb(1, x_2) = 0 + 0 = 0$ and $LB(0) = \delta(0) + lb(0, x_2) = 0 + 1 = 1$. Since $LB(1) < LB(0)$, we have $LB = LB(1) = 0$. Similarly, $UB = Inf$. Since $threshold = 0$ is not equal $UB = Inf$, the test in line 38 fails. The test in line 40 succeeds and x_1 will choose its value d that minimizes $LB(d)$. Thus, x_1 switches value to $x_1 = 1$. It will again send VALUE messages to its linked descendents (figure 6.c).

Next, let us assume that the COST messages sent to x_2 in figure 6.b are delayed. Instead, x_2 receives x_1's VALUE message from figure 6.c. In line 15, x_2 will update its $CurrentContext$ to $\{(x_1, 1)\}$. For brevity, the remaining portion of this procedure is not described.

Next, x_2 finally receives the COST message sent to it from x_3 in figure 6.b. x_2 will test the received context against $CurrentContext$ and find that they are incompatible because one contains $(x_1, 0)$ while the other contains $(x_1, 1)$ (line 31). Thus, the costs in that COST message will not be stored due to the context change. However, the COST message from x_4 will be stored in lines 32-33 as $lb(0, x_3) = 1$ and $ub(0, x_3) = 1$. In line 37, x_2 then proceeds to the **backTrack** procedure where it will choose its best value. The best value is now $x_2 = 1$ since $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 0 + 0 + 0$ and $LB(0) = \delta(0) + lb(0, x_3) + lb(0, x_4) = 2 + 0 + 1 = 3$. Figure 6.d shows the change in both x_2 and x_3 values after receiving x_1's VALUE message from figure 6.c. x_2 and x_3 send the new COST messages with the new context where $x_1 = 1$. x_2 also sends VALUE messages to x_3 and x_4 informing them of its new value.

Next, figure 6.e shows the new COST message that is sent by x_2 to x_1 after receiving the COST messages sent from x_3 and x_4 in figure 6.d. Notice that x_2 computes LB as $LB(1) = \delta(1) + lb(1, x_3) + lb(1, x_4) = 0 + 0 + 0$ and UB as $UB(1) = \delta(1) + ub(1, x_3) + ub(1, x_4) = 0 + 2 + 1 = 3$. Figure 6.e also shows the new COST message sent by x_3 after receiving x_2's new value of $x_2 = 1$. Similarly, x_4 will change variable value and send a COST message with $LB = 0$ and $UB = 0$. In this way, we see the agents have ultimately settled on the optimal configuration with all values equal to 1 (total cost = 0).

Finally in figure 6.f, x_2 receives the COST messages from figure 6.e, computes a new bound interval $LB = 0, UB = 0$ and sends this information to x_1. Upon receipt of this message, x_1 will compute $UB = UB(0) = \delta(0) + ub(0, x_2) = 0 + 0 = 0$. Note that x_1's $threshold$ value

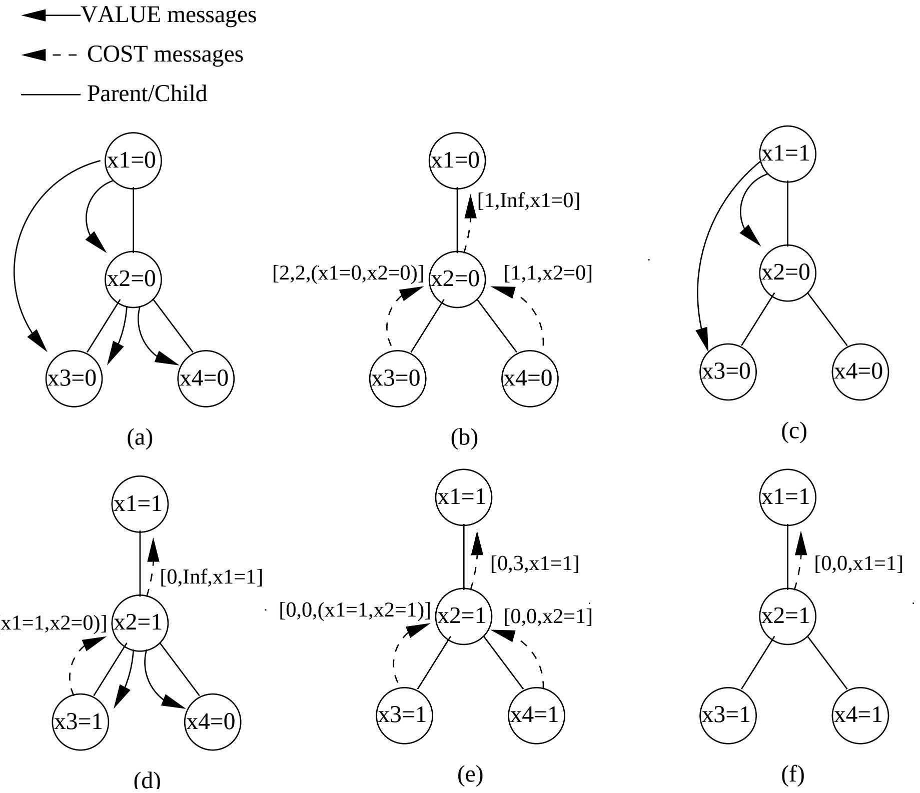


Figure 6: Example Adopt execution for the DCOP shown in figure 3

is also equal to zero. $threshold$ was initialized to zero in line 1 and can only be increased if i) a THRESHOLD message is received (line 8), or b) the ThresholdInvariant is violated (line 54, figure 5). The root never receives THRESHOLD messages, so case (i) never occurred. Since x_1's LB was never greater than zero in this example, $threshold$ could never have been less than LB, so case (ii) never occurred. Thus, $threshold$ was never increased and remains equal to zero. We have the test $threshold == UB$ in line 45 evaluate to true. In line 48, it will send a TERMINATE message to x_2, and then x_1 will terminate in line 51. x_2 will receive the TERMINATE message in line 11, evaluate $threshold == UB(= 0)$ to be true in line 45 and then terminate in line 51. The other agents will terminate in a similar manner.

4.3 Example of Backtrack Thresholds

We illustrate how backtrack thresholds are computed, updated and balanced between children. The key difficulty is due to context changes. An agent only stores cost information for the current context. When the context changes, the stored cost information must be deleted (in order to maintain polynomial space). If a previous context is later returned to, the agent no longer has the previous context's detailed cost information available. However, the agent had reported the total sum of costs to its parent, who has that information stored. Although the precise information about how the costs were accumulated from the children is lost, the total sum is available from the parent. It is precisly this sum that the parent sends to the agent via the THRESHOLD message. The child then heuristically re-subdivides, or allocates, the threshold among its own children. Since this allocation may be incorrect, it then corrects for over-estimates over time as cost feedback is (re)received from the children.

Figure 7 shows a portion of a DFS tree. The constraints are not shown. Line numbers mentioned in the description refer to figure 4 and figure 5. x_p has parent x_q, which is the root, and two children x_i and x_j. For simplicity, assume $D_p = \{d_p\}$ and $\delta(d_p) = 1$, i.e, x_p has only one value in its domain and this value has a local cost of 1.

Suppose x_p receives COST messages containing lower bounds of 4 and 6 from its two children (figure 7.a). The costs reported to x_p are stored as $lb(d_p, x_i) = 4$ and $lb(d_p, x_j) = 6$ (line 32) and associated context as $context(d_p, x_i) = context(d_p, x_j) = \{(x_q, d_q)\}$. LB is computed as $LB = LB(d_p) = \delta(d_p) + lb(d_p, x_i) + lb(d_p, x_j) = 1 + 4 + 6 = 11$. In figure 7.b, the corresponding COST message is sent to parent x_q. After the COST message is sent, suppose a context change occurs at x_p through the receipt of a VALUE message $x_q = d'_q$. In line 18-19, x_p will reset $lb(d_p, x_i)$, $lb(d_p, x_j)$, $t(d_p, x_i)$ and $t(d_p, x_j)$ to zero.

Next, x_q receives the information sent by x_p. x_q will set $lb(d_q, x_p) = 11$ (line 32), and enter the **maintainChildThresholdInvariant** procedure (line 35). Let us assume that $t(d_q, x_p)$ is still zero from initialization. Then, the test in line 65 succeeds since $lb(d_q, x_p) = 11 > t(d_q, x_p) = 0$ and x_q detects that the ChildThresholdInvariant is being violated. In order to correct this, x_q increases $t(d_q, x_p)$ to 11 in line 66.

Next, in figure 7.c, x_q revisits the value d_q and sends the corresponding VALUE message $x_q = d_q$. Note that this solution context has already been explored in the past, but x_p has retained no information about it. However, the parent x_q has retained the sum of the costs, so x_q sends the THRESHOLD message with $t(d_q, x_p) = 11$.

Next, x_p receives the THRESHOLD message. In line 8, the value is stored in the $threshold$ variable. Execution proceeds to the **backTrack** procedure where **maintainAllocationInvariant** is invoked in line 44. Notice that the test in line 57 of **maintainAllocationInvariant** evaluates to true since $threshold = 11 > \delta(d_p) + t(d_p, x_i) + t(d_p, x_j) = 1 + 0 + 0$. Thus, in lines 57-59, x_p increases the thresholds for its children until the invariant is satisfied. Suppose that the split is $t(d_p, x_i) = 10$ and $t(d_p, x_j) = 0$. This is an arbitrary subdivision that satisfies the AllocationInvariant since there are many other values of $t(d_p, x_i)$ and $t(d_p, x_j)$ that could be used. In line 63, these values are sent via a THRESHOLD message (figure 7.d).

By giving x_i a threshold of 10, x_p risks sub-optimality by overestimating the threshold in that subtree. This is because the best known lower bound in x_i's subtree was only 4. We now show how this arbitrary allocation of threshold can be corrected over time. Agents continue execution until, in figure 7.e, x_p receives a COST message from its right child x_j indicating that the lower bound in that subtree is 6. x_j is guaranteed to send such a message because there can be no solution in that subtree of cost less than 6, as evidenced by the COST message previously sent by x_j in figure 7.a. x_p will set $lb(d_p, x_j) = 6$ (line 32) and enter the **MaintainChildThresholdInvariant** procedure in line 35. Note that the test in line 65 will succeed since $lb(d_p, x_j) = 6 > t(d_p, x_j) = 5$ and the ChildThresholdInvariant is being violated. In order to correct this, x_p increases $t(d_p, x_j)$ to 6 in line 66. Execution returns to line 35 and continues to line 44, where the **maintainAllocationInvariant** is invoked. The test in line 60 of this procedure will succeed since $threshold = 11 < \delta(d_p) + t(d_p, x_i) + t(d_p, x_j) = 1 + 10 + 6 = 17$ and so the AllocationInvariant is being violated. In lines 60-62, x_p lowers $t(d_p, x_i)$ to 4 to satisfy the invariant. In line 63, x_p sends the new (correct) threshold

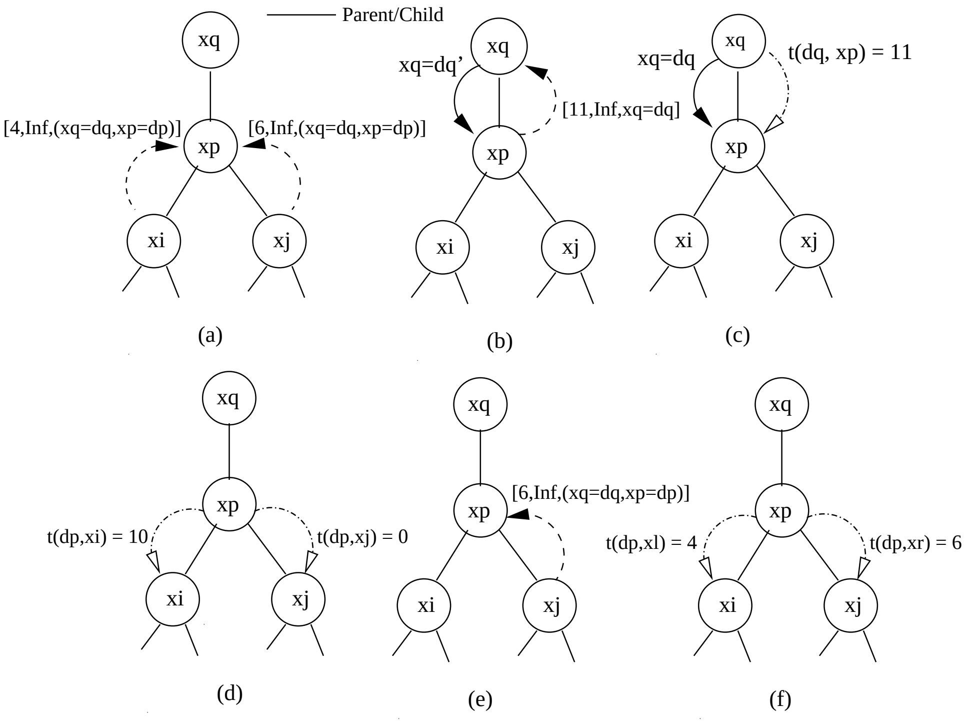


Figure 7: Example of backtrack thresholds in Adopt

values to its children (figure 7.f).

In this way, a parent agent continually rebalances the threshold given to its independent subtrees in order to avoid overestimating the cost in each subtree and allowing more efficient search.

4.4 Evaluation

As in previous experimental set-ups[7], we experiment on distributed graph coloring with 3 colors. One node is assigned to one agent who is responsible for choosing its color. Cost of solution is measured by the total number of violated constraints. We will experiment with graphs of varying *link density*. A graph with link density d has dn links, where n is the number of nodes in the graph. For statistical signficance, each datapoint representing number of cycles is the average over

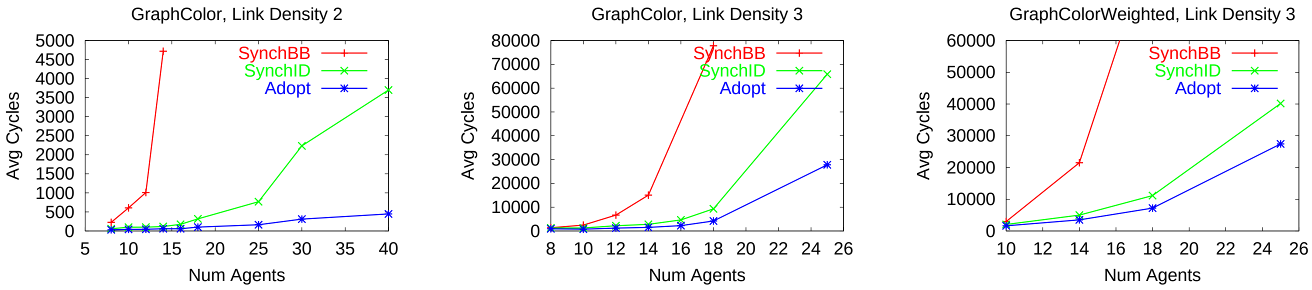


Figure 8: Average number of cycles required to find the optimal solution

25 random problem instances. The randomly generated instances were not explicitly made to be overconstrained, but note that link density 3 is beyond phase transition, so randomly generated graphs with this link density are almost always overconstrained. Also as in [7], time to solution is measured in terms of synchronous cycles. One *cycle* is defined as all agents receiving all their incoming messages and sending out all their outgoing messages. The tree-structured DFS priority ordering for Adopt was formed in a preprocessing step. To compare Adopt's performance with algorithms that require a chain (linear) priority ordering, a depth-first traversal of Adopt's DFS tree was used.

4.5 Efficiency

We present the empirical results from experiments using three different algorithms for DCOP: Synchronous Branch and Bound (SynchBB), Synchronous Iterative Deepening (SynchID) and Adopt. We illustrate that Adopt outperforms SynchBB[6], a distributed version of branch and bound search and the only known algorithm for DCOP that provides optimality guarantees. In addition, by comparing with SynchID we show that the speed-up comes from two sources: a) Adopt's novel search strategy, which uses lower bounds instead of upper bounds to do backtracking, and b) the asynchrony of the algorithm, which enables concurrency.

SynchID is an algorithm we have constructed in order to isolate the causes of speed-ups obtained by Adopt. SynchID simulates iterative deepening search[10] in a distributed environment. SynchID's search strategy is similar to Adopt since both algorithms iteratively increase lower bounds and use the lower bounds to do backtracking. The difference is that SynchID maintains a single global lower bound and agents are required to execute sequentially and synchronously, while in Adopt, each agent maintains its own lower bound and agents are able to execute concurrently and asynchronously.

Figure 8 shows how SynchBB, SynchID and Adopt scale up with increasing number of agents on graph coloring problems. The results in Figure 8 (left) show that Adopt significantly outperforms both SynchBB and SynchID on graph coloring problems of link density 2. The speed-up of Adopt over SynchBB is 100-fold at 14 agents. The speed-up of Adopt over SynchID is 7-fold at 25 agents and 8-fold at 40 agents. The speedups due to search strategy are significant for this problem class, as exhibited by the difference in scale-up between SynchBB and SynchID. In addition, the figure also shows the speedup due exclusively to the asynchrony of the Adopt algorithm. This is exhibited by the difference between SynchID and Adopt, which employ a similar search strategy, but differ in amount of asynchrony. In SynchID, only one agent executes at a time so it has no asynchrony, whereas Adopt exploits asynchrony when possible by allowing agents to choose variable values in parallel. In summary, we conclude that Adopt is significantly more effective than SynchBB on sparse constraint graphs and the speed-up is due to both its search strategy and its exploitation of asynchronous processing. Adopt is able to find optimal solutions very efficiently for large problems of 40 agents or more.

Figure 8 (middle) shows the same experiment as above, but for denser graphs, with link density 3. We see that Adopt still outperforms SynchBB around 10-fold at 14 agents and at least 18-fold at 18 agents (experiments were terminated after 100000 cycles). The speed-up between Adopt and SynchID, i.e, the speed-up due to concurrency, is 2.06 at 16 agents, 2.22 at 18 agents and 2.37 at 25 agents. Finally, Figure 8 (right) shows results from a weighted version of graph coloring where each constraint is randomly assigned a weight between 1 and 10. Cost of solution is measured as the sum of the weights of the violated constraints. We see similiar results on the more general problem with weighted constraints.

These experiments demonstrate that although distributed constraint optimization is intractable in the worst case, some classes of problems exhibit special properties in which optimal algorithms can perform very well. In particular, Adopt is able to guarantee optimality at low cost for large problems when the constraint network is sparse, a typical feature of distributed sensor networks [14].

5 Application of DCR to Distributed Sensor Networks

This section presents techniques to apply DCR algorithms to real-world hardware. First, we have developed a two-layered architecture. The general principle is to allow a coordination algorithm to solve the problem for which it was designed by encapsulating algorithms for dealing with hardware issues into a lower layer. The lower layer is a probabilistic component that deals with task uncertainty and dynamics and allows an agent to do local reasoning when time for coordination is not available. The lower layer provides information to the higher layer about which tasks are present, thus presenting the DCR algorithm with an abstracted problem. DCR runs as the higher layer coordinating inter-agent activities. The higher layer, which is able to communicate with other agents, provides non-local information about the presence of tasks to the lower layer thereby allowing the lower layer to update its probability model about which tasks are present. The incorporation of non-local information gives each sensor a more accurate picture of the environment and results in better overall system performance.

Since agents cannot determine with certainty which tasks must be performed, they must represent the probability that a particular task is present. The second key idea is to update the probability distribution using both information from the agent's sensors and using information inferred from the distributed constraint problem-solving between agents. The dynamics of the environment are handled by continually updating the probability distribution and informing the DCR algorithm when "significant" changes occur. DCR allocates resources to tasks with a high probability of being present. While DCR has the capability to make (probably sub-optimal) intermediate solutions available while it continues to search for an optimal solution, it can potentially take some time to find even a first solution once the situation changes. The third key idea is that the lower layer provides an ability to make fast, local resource allocation decisions when time for coordination is limited. This ensures reasonable operation in a dynamic real-time environment. Thus, the lower layer can perform a simple allocation of resources while DCR works on finding a good global allocation of resources.

We present results obtained from an implementation of the system described above on hardware sensors. The results show that the two-layered architecture is effective at tracking moving targets. The probability model and local reasoning enabled the multiagent algorithm to deal with the difficulties posed by the distributed sensor domain, allowing us to incorporate an existing "off the shelf" distributed constraint reasoning algorithm into a real application. As far as we are aware, this is the first report of successful application of DCR on real hardware. We believe this is a significant first step towards moving multiagent algorithms developed on abstract problems into robotic applications.

5.1 Distributed Constraint Reasoning for Distributed Sensors

To use DCR in a distributed sensor domain it was necessary to add a second layer which maintains a probabilistic representation of the currently present tasks. This layer presents an abstracted problem to DCR which allows the multiagent algorithm to perform its core task of finding good solutions to an abstract problem. The probabilistic layer also performs resource allocation itself, when DCR does not have solutions immediately available. In the remainder of this section, we describe the overall architecture, including the interaction between the two layers. In the next section we describe the details of the probabilistic layer in more detail.

The probabilistic layer has two distinct modes of operation. Which mode of operation to use is determined by whether DCR has selected allocated resources to tasks that are currently present. In the sensor network domain this can be determined by whether DCR has specified using a radar head that can detect a target. If DCR has selected a currently present task for the node to work on, the probabilistic layer has a passive monitoring role. If DCR allocates the agent to a task that is not present then the probabilistic layer acts pro-actively to determine which tasks are present and allocate resources to one of the tasks. In other words, the probabilistic layer is pro-active in making resource allocation decisions when DCR does not have an up-to-date resource allocation. DCR is given responsibility for making decisions if it can because it is able to make globally optimal allocation decisions while the probabilistic layer can only randomly choose between detected targets. Notice, that the probabilistic layer does its reasoning asynchronously and in parallel to DCR's resource allocation reasoning.

In its passive monitoring mode, the probabilistic layer updates its probability distribution and informs DCR when the status of a task changes, e.g., when a task status changes from present to not present. In its active mode, the probabilistic layer takes actions that will indicate the presence of a task as quickly as possible. Determining which action will most readily indicate the presence of a task can be inferred from the sensor model (see below). The action is determined by calculating which action has the highest probability of giving a reading that will reduce uncertainty about the presence of tasks.

Figure 9 shows the channels of communication and the information that flows along those channels in an agent. Notice that information flows down from the high level multiagent negotiation and up from the low level probabilistic layer. This allows the agent to take advantage of both local information, i.e., sensor readings, and global information, i.e., inferred information from other agents, giving it an accurate picture of which tasks are present. Information flows up from the probabilistic representation to DCR via messages indicating that the status of a task has changed. For example, a message is sent when the status of a task changes from U to P. Information flows down, from DCR to the probabilistic layer indicating which tasks other agents believe to be present, whenever it receives a communication providing that information. For example, if Agent 2 communicates the presence of Task 1 to Agent 1, DCR at Agent 1 will send a message to the probabilistic layer indicating the presence of Task 1.

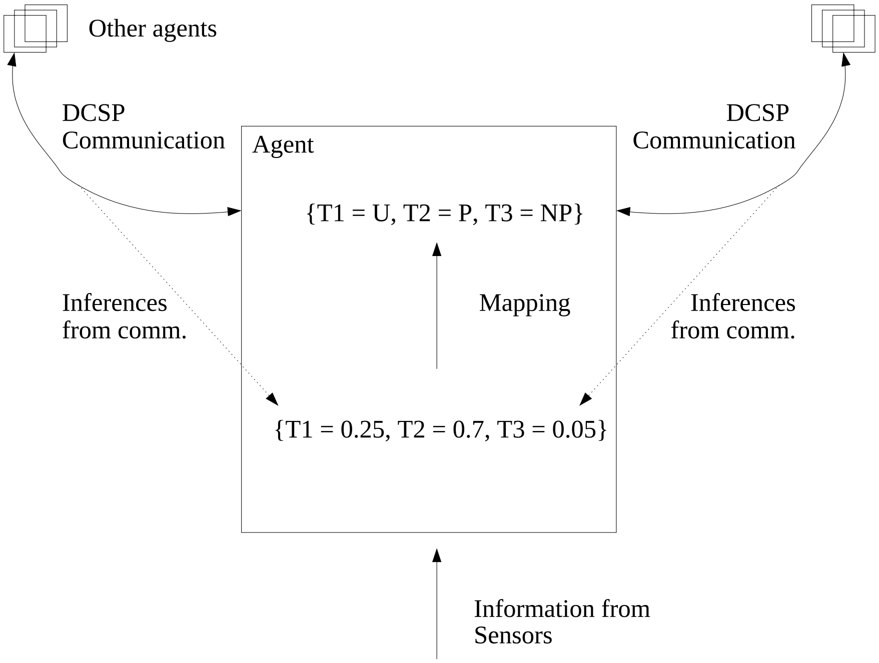


Figure 9: Diagram of the basic information flows around an agent.

5.2 Probabilistic Task Representation

For each task in T_a, a task status from $\{P, NP, U\}$, representing present, not present and unknown respectively, is maintained. The aim of the uncertainty reasoning is to have the agent have the value P for each task in T_p. Tasks, which map to variables in DCR, with status P should be assigned either value $Allocated$ or $Ignored$ by DCR. Tasks with status NP should be assigned the value $NotPresent$ by DCR. Finally, tasks with status U can be assigned any value, depending on the value at other agents by DCR. For these tasks, DCR relies on task information from provided by agents to make its local decisions.

For each $T \in T_a$ the probability that the task is currently present is $Pr(T)$. The agent maintains this probability for each task, i.e., it maintains probabilities $\mathbf{PR} = \{Pr(T_1) \ldots Pr(T_N)\}$. Each agent's probability distribution is maintained locally, hence different agents may have different probabilities that a task is present. A function maps probabilities to task status values. In the sensor network domain we use the following mapping: **if** $Pr(T) < 0.2$ **then** NP **else if** $Pr(T) > 0.8$ **then** P **else** U. The thresholds for deciding whether a tasks is present, not present or unknown are determined experimentally.

Maintaining as accurate as possible probability distribution, and thus, giving DCR as accurate as possible set of present tasks, is essential to the success of this approach. To create and maintain this distribution in a noisy, dynamic environment requires the combination of multiple measurements to reduce uncertainty. While old information can be useful, more weight is given to the most recent measurements since the environment, i.e., the present tasks, changes dynamically. Four pieces of information are used to update the probability distribution.

- Updates based on observations made while performing a task, using a learned environment model. Formally, this information is $Pr(T|S)$, where S is a sensor reading.

- Updates made based on inferences from overheard communications from other nodes. Formally, this information is $Pr(T|M)$ where M is a message.
- Updates made based on knowledge of the dynamics of the domain. In particular, the probability that a task is present given the probability that it was present earlier. Formally, this information is $Pr(T_t|T_{t-1})$, where $Pr(T_t)$ is the probability task T is present at time t.

We refer to each type of information as an observation, denoted O. Each of the types of observation provides some evidence about the presence of a task, T. In particular, given a model of the types of observation that can be received we can calculate $Pr(T|O)$. That evidence should be combined with previous evidence to make $Pr(T)$ more accurate. However, since the situation changes dynamically, more recent evidence should be weighted more heavily than older information. The integration of the new observations with the previous evidence uses a variation on Bayes' rule:

$$Pr(T|O) = \frac{Pr(O|T) \times Pr(T)}{Pr(O)}$$

In this equation O is the new observation. $Pr(O|T)$ is the probability of getting the observation given that the task is present. This probability is calculated in different ways, depending on the type of observation. For example, a model of the sensors provides this information for sensor observations. $Pr(T)$ is the a priori probability of task T. Since we know the probability that the task was present in the previous time step we can use that information to calculate the probability that the task is present in the current time step. That is:

$$Pr(T_t) = \frac{Pr(T_t|T_{t-1}) \times Pr(T_{t-1})}{Pr(T_{t-1}|T_t)}$$

where $Pr(T_t)$ is the probability of task T being present at time t. For simplicity, we set $\frac{Pr(T_t|T_{t-1})}{Pr(T_{t-1}|T_t)} = w$. Essentially, this assumes that the dynamics of the environment are uniform across tasks and times. Thus, the calculation of the probability of a task given a new measurement and an previous probability is:

$$Pr(T_t|O) = \frac{Pr(O|T_t) \times wPr(T_{t-1})}{Pr(O)}$$

The integration of new observations iteratively updates $\mathbf{PR}$. When any $Pr(T)$changes enough that it causes the status of a task to change, e.g., NP to P, a message is sent to DCR which then may start a new round of negotiation to determine a new optimal task allocation.

DCR assigns weights to tasks, prioritizing tasks with higher weights. Normally, if the status of some task is U the agent will not actively try to allocate resources to that task, nor will it take actions to determine whether or not the task is actually present. However, if it is currently allocating resources to a task with lower weight than a task with status U it will periodically schedule actions to resolve the uncertainty surrounding that task. In particular, in the sensor network domain it can switch to the sector most likely to determine whether or not the task is present. This behavior ensures that important tasks are not ignored because no agent checks whether the task is present. However, agents do not spend time checking for tasks that are of lower priority than the one to which they are currently allocating resources.

	Task							
Reading	T1	T2	T3	T4	T5	T6	T7	T8
0	0.83	0.69	0.65	0.51	0.39	0.39	0.78	0.89
1	0.16	0.29	0.28	0.27	0.19	0.15	0.15	0.10
2	0.0	0.01	0.05	0.17	0.17	0.14	0.03	0.0
3	0.0	0.0	0.0	0.03	0.15	0.11	0.02	0.0
4	0.0	0.0	0.0	0.0	0.05	0.05	0.0	0.0
5	0.0	0.0	0.0	0.0	0.01	0.13	0.0	0.0

Table 2: Part of the model of $Pr(O|T)$ for one sensor and sector and a subset of tasks.

Strength	0	1	2	3	4	5
Probability	0.83	0.09	0.02	0.01	0.01	0.02

Table 3: $Pr(O)$ for a particular sensor and sector.

5.3 Updates from Sensors

With a model of its environment and readings from it sensors, an agent can apply Bayes' rule to reason about the presence of all possible tasks. Such a technique for reducing uncertainty is purely local, i.e., only local information is used. To apply Bayes' rule the agent needs to know $Pr(O|T)$ and $Pr(O)$. Table 2 shows part of the model of $Pr(O|T)$, i.e., a model of the probability of a reading given the presence of a task, for a subset of tasks for a particular node and sector. Column 1 gives the strength of the reading, with higher numbers representing stronger readings. Readings of strength 0 and 1 cannot be distinguished from noise. Columns 2-9 show the probability of getting a reading of that strength given that the task is currently present. This sensor can give little information about the presence of Task 1, since even if the task is present the readings will be no stronger than noise. On the other hand, for Task 6, the sensor will get a reading of strength 5 13% of the time, if Task 6 is present. Table 3 shows the a priori probability of getting readings of various strengths. The table shows that readings of strength 5 are quite rare. Using Bayes' rule and the initial probability of Task 6 being present, a reading of strength 5 would markedly increase the probability of Task 6 being present. For the distributed sensor domain, $Pr(O|T)$ can be calculated analytically from a model of the sensor.

5.4 Updates from Overheard Communication

In order to find a good allocation of resources to tasks, DCR requires agents negotiate as described above. Since the probability distribution of each agent will be different and the local sensing actions of each agent are suited to detecting particular tasks, agents can infer useful information from communications from other agents. Inferring information from communication is a non-local method for reducing uncertainty about which tasks are currently present.

DCR negotiations do not deal with the probabilities, instead they use the abstracted $Ignore$, $Allocated$ and $NotPresent$ values. Messages with $Ignore$ or $Allocated$ imply that the sender agent believes the task is present, while messages with $NotPresent$ imply that the agent believes the task is absent. Since each agent uses the same probabilistic reasoning, a receiving agent knows that an agent that sent a message indicating the presence (absence) of a task then the sending agent

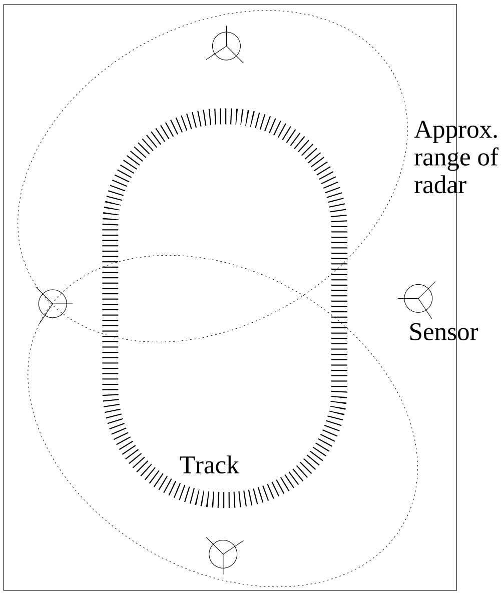


Figure 10: The configuration of the room, sensors and target track for hardware experiments. Dotted ellipses from sensor on left hand sensor show approximate range of two of the sensors radar heads.

must have a probability above (below) its threshold for that task. However, because the sending agent may send a message about a task even though the probability that task is present is less than 1.0, the receiving agent cannot assume the task is definitely present and must apply Bayes' rule when updating its own probability distribution based on that information.

5.5 Hardware Experiments

Below, we describe the results of two separate illustrative experiments. In the first experiment, the sensors were arranged in a diamond formation. The experiment was setup in a small room in an office environment which provided a very noisy environment for the sensors. The configuration is shown in Figure 10. The lines on the sensors show the orientation of the radar heads. Notice that the sensors at the ends of the room did not need to change sectors, while the sensors on the sides of the room needed to switch between two sectors.

The aim of the sensor network is to obtain an accurate track of one or more moving targets. Creating such a track involves a variety of algorithms working together, e.g., the task allocation algorithm and an algorithm for combining measurements from multiple sensors. A sample track is shown in Figure 12. The quality of the track is dependent on the number of measurements taken. The more the measurements, the better the results. To check if competing algorithms might have performed better, we present the results from three different algorithms in Figure 11 (the x-axis shows the number of measurements taken and the y-axis shows the algorithm used). The first algorithm used a fixed configuration of sectors based on the known track of the target. One configuration had the sensors on the sides of the room both looking towards one end of the room ("fixed up" in the figure), while the other had the sensors on the sides of the room looking to opposite ends of the room ("fixed u/d" in the figure). The next algorithm (”local” in the figure)

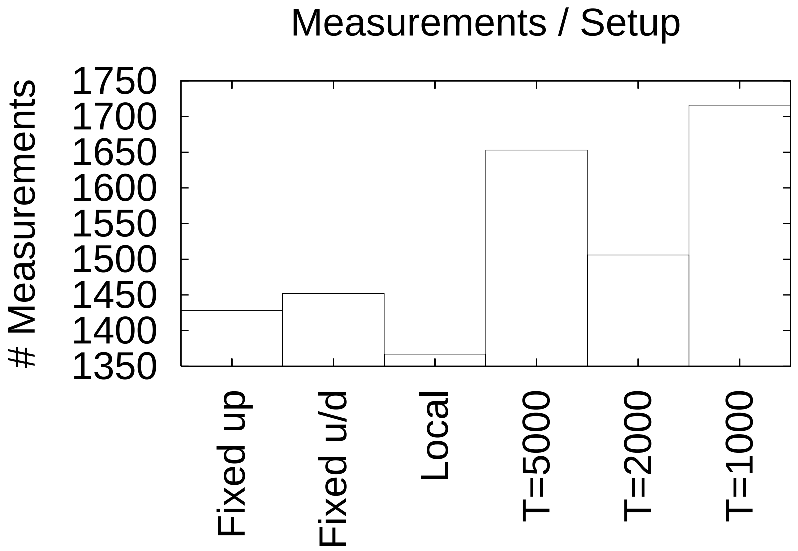


Figure 11: Number of measurements made by various algorithms.

used only local sensing information, changing sectors whenever it failed to sense a target in the sector it was currently using. Finally, DCR was used with various timeout lengths (1 second – "T=1000" in the figure, 2 seconds – "T=2000" and 5 seconds – "T=5000"). Each algorithm was run three times, each time for 20 minutes. The values shown on the graphs are the average number of measurements across the three runs.

DCR performed clearly better than the other algorithms because the four nodes together were better able to resolve uncertainty and find the target than the localized algorithms. The "local" algorithm performed worst because it was most susceptible to the noise in the environment. A single false reading indicating the presence of a target would result in the agent wasting a significant amount of time. The algorithms utilizing information from others as well as their own information were less susceptible to single noisy measurements. The reason for the difference in performance of DCR is related to the ratio of the speed of the moving target and the speed at which nodes can communicate.

Figure 13 shows a track produced using the two layered technique with a software simulator. The simulator accurately captures the noise and dynamics experienced in the real domain. In this case there were eight nodes and two trains. The figure shows the track produced for one of the trains, which moved on an oval shaped track around a room. To be tracking both trains at once at least six of the nodes needed to be taking measurements. Often, a node could take measurements of both targets and negotiation was required to determine which target it should track.

6 Efficient DCR Reasoning

This section presents efficient DCR techniques to enable fast convergence to a solution in conflict resolution. Fast convergence is critical in many multiagent applications under real-time constraints. For instance, in distributed sensor networks, agents must resolve conflicts over shared sensors as

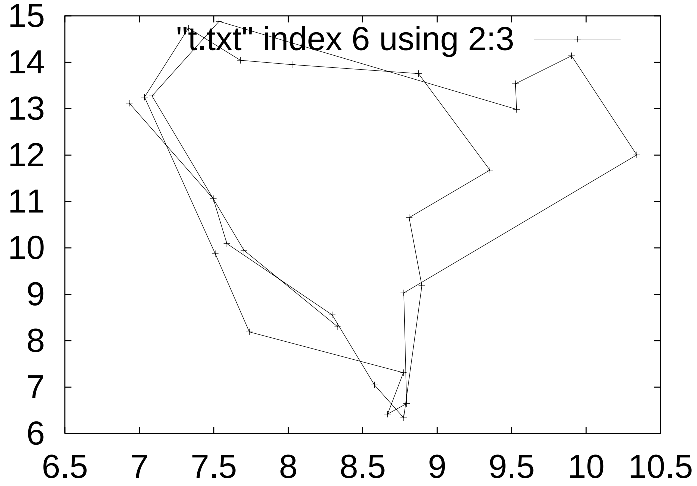


Figure 12: Track produced by four hardware sensors following a target moving on an oval track.

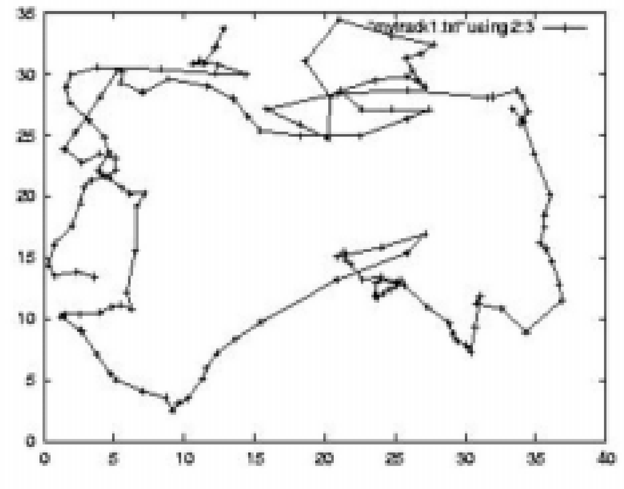


Figure 13: Track of target moving on an oval shaped track.

quickly as possible since the delay in conflict resolution may lead to a situation where no target is tracked before moving out of sensor range. Here, we present new DCR strategies for conflict resolution and show systematic investigation of their performance particularly in a large scale multiagent system. For experimental purpose, the new strategies are incorporated into the best published DCSP algorithm called Asynchronous Weak Commitment (AWC) search algorithm.

6.1 Asynchronous Weak Commitment (AWC) Search Algorithm

Asynchronous Weak Commitment (AWC) search algorithm is the best published DCSP algorithm[22]. In the AWC approach, agents asynchronously assign values to their variables from domains of possible values, and communicating the values to neighboring agents with shared constraints. Each variable has a non-negative integer priority that changes dynamically during search. A variable is consistent if its value does not violate any constraints with higher priority variables. A solution is a value assignment in which every variable is consistent.

To simplify the description of the algorithm, suppose that each agent has exactly one variable and the constraints between variables are binary. When the value of an agent's variable is not consistent with the values of its neighboring agents' variables, there can be two cases: (i) a *good* case where there exists a consistent value in the variable's domain; (ii) a *nogood* case that lacks a consistent value. In the good case with one or more value choices available, an agent selects a value that minimizes the number of conflicts with lower priority agents. In the nogood case, an agent increases its priority to *max+1,* where *max* is the highest priority of its neighboring agents, and selects a new value that minimizes the number of conflicts with all of its neighboring agents. This priority increase makes previously higher agents select new values. Agents avoid the infinite cycle of selecting non-solution values by saving the *nogood* situations.

6.2 Cooperativeness-based Strategies

While AWC is one of the most efficient DCSP algorithms, real-time and dynamism in multiagent domains demands very fast conflict resolution convergence. We introduce new strategies which consider how much flexibility (choice of values) is given towards other agents by a selected value. By considering neighboring agents' local constraints, an agent can generate a more locally cooperative response, potentially leading to faster conflict resolution convergence. Note that AWC relies on the *min-conflict* heuristic [13] that minimizes conflicts with other agents.

The concept of local cooperativeness goes beyond merely satisfying constraints of neighboring agents to accelerate convergence. That is, an agent A_i cooperates with a neighbor agent A_j by selecting a value for its variable that not only satisfies the constraint with A_j, but also maximizes A_j's flexibility (choice of values). Then A_j has more choices for a value that satisfies A_j's local constraints and other external constraints with its neighboring agents, which can lead to faster convergence. To elaborate this notion of local cooperativeness, the followings were defined.

- **Definition 1**: For a value $v \in D_i$ and a set of agents $N_i^{sub} \subseteq N_i$ (a set of neighboring agents), a *flexibility function* is defined as $f(v, N_i^{sub}) = \Sigma_j c(v, A_j)$ such that $A_j \in N_i^{sub}$ and $c(v, A_j)$ is the number of values of A_j that are consistent with v.

- **Definition 2**: For a value v of A_i, *local cooperativeness* of v is defined as $f(v, N_i)$. That is, the *local cooperativeness* of v measures how much flexibility (choice of values) is given to all of A_i's neighbors by v.

- **Definition 3**: A *maximally cooperative* value of X_i is defined as v_{max} such that, for any other value $v_{other} \in D_i$, $f(v_{max}, N_i) \geq f(v_{other}, N_i)$.

Here, the concept of cooperativeness goes beyond merely satisfying constraints of neighboring agents and enables even faster convergence. That is, an agent A_i can provide a more cooperative response to a neighbor agent A_j, by selecting a value that not only satisfies the constraint with A_j, but maximizes flexibility (choice of values) for A_j. If A_i selects v_{max}, giving A_j more choice, then A_j can more easily select a value that satisfies A_j's local constraints and other external constraints with its neighboring agents such as A_k. Having lower possibility of constraint violation, this cooperative response can lead to faster convergence.

As an example of the flexibility function $f(v, N_i^{sub})$, suppose agent A_1 has two neighboring agents A_2 and A_3, where a value v leaves 70 consistent values to A_2 and 40 to A_3 while another value v' leaves 50 consistent values for A_2 and 49 to A_3. Now, assuming that values are ranked based on flexibility, an agent will prefer v to v': $f(v, \{A_2, A_3\})$ = 110 and $f(v', \{A_2, A_3\})$ = 99. These definitions of flexibility function and local cooperativeness are applied for the cooperative strategies defined as follows:

- S_{basic}: Each agent A_i selects a value based on min-conflict heuristics (the original strategy in the AWC algorithm);

- S_{high}: Each agent A_i attempts to give maximum flexibility towards its higher priority neighbors by selecting a value v that maximizes $f(v, N_i^{high})$;

- S_{low}: Each agent A_i attempts to give maximum flexibility towards its lower priority neighbors by selecting a value v that maximizes $f(v, N_i^{low})$;

- S_{all}: Each agent A_i selects a value v that maximizes $f(v, N_i)$, i.e. max flexibility to all neighbors.

We now define cooperativeness relation among these strategies based on the cooperativeness of values they select.

- **Definition 4**: For two different strategies S_α and S_β, S_α is *more cooperative* than S_β iff (i) for all A_i, v_α, $v_\beta \in D_i$ such that v_α and v_β are selected by S_α and S_β respectively, $f(v_\alpha, N_i)$ $\geq f(v_\beta, N_i)$ and (ii) for some A_i, when $f(v_\alpha, N_i) \neq f(v_\beta, N_i)$, $f(v_\alpha, N_i) > f(v_\beta, N_i)$.

- **Theorem 1**: The strategy S_{all} is *maximally cooperative* strategy in the *good* case, i.e., for any other strategy S_{other}, S_{all} is more cooperative than S_{other}.
  Proof: By contradiction. Assume that S_{other} is more cooperative. For A_i, v_{all} is selected by S_{all} and v_{other} by S_{other} such that if $f(v_{all}, N_i) \neq f(v_{other}, N_i)$, then $f(v_{all}, N_i) < f(v_{other}, N_i)$. However, by the definition of S_{all}, v_{other} would be selected by S_{all} instead of v_{all}. This is a contradiction.

The four strategies defined above can be applied to both the *good* and *nogood* cases. Therefore, there are sixteen strategy combinations for each flexibility base. Since, we will only consider strategy combinations, henceforth, we will refer to them as strategies for short. Note that all the strategies are enhanced with constraint communication and propagation. Here, two exemplar strategies are listed.

- $S_{basic} - S_{basic}$: This is the original AWC strategy. Min-conflict heuristic is used for the good and nogood cases.
- $S_{low} - S_{high}$: For the good case, an agent is most locally cooperative towards its lower priority neighbor agents by using S_{low}. (Note that the selected value doesn't violate the constraints with higher neighbors). On the contrary, for the nogood situations, an agent attempts to be most locally cooperative towards its higher priority neighbors by using S_{high}.
- **S$_{all}$-S$_{all}$**: In both the *good* and the *nogood* cases, an agent uses S$_{all}$ which is to select a value that maximizes flexibility of all neighbor agents.

By theorem 1, S$_{all}$ is more cooperative than the other strategies S$_{high}$, S$_{low}$, S$_{basic}$ for the *good* case. Both S$_{high}$ and S$_{low}$ have trade-offs. For instance, S$_{high}$ may leave very little or no choice to an agent's neighbors in N$_i^{low}$, making it impossible for them to select any value for the neighbors. S$_{low}$ has a converse effect. S$_{basic}$ also has trade-offs because it does not consider the flexibility of neighboring agents.

S$_{basic}$ tries to minimize the number of constraint violations without taking neighboring agents' own restrictions into account. An agent A_i's selected value v with S$_{basic}$ is thus not guaranteed to be maximally cooperative, i.e., $f(v, N_i) \leq f(v_{max}, N_i)$. Hence, S$_{basic}$ is not the most cooperative strategy to neighboring agents.

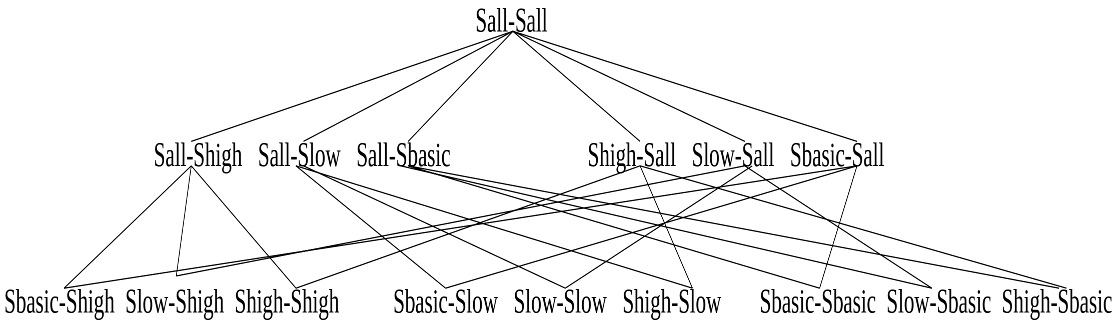


Figure 14: Cooperativeness relationship

6.3 Experimental Evaluation

An initial principled investigation of these strategies can improve our understanding not only of DCSP strategies, but potentially shed some light on how cooperative an agent ought to be towards its neighbors, and with which neighbors. To that end, two different types of DCSP configurations were considered in the experiments: a chain and a grid. In the chain configuration, each agent had two neighboring agents, to its right and left (except for end points). Since there was a constraint

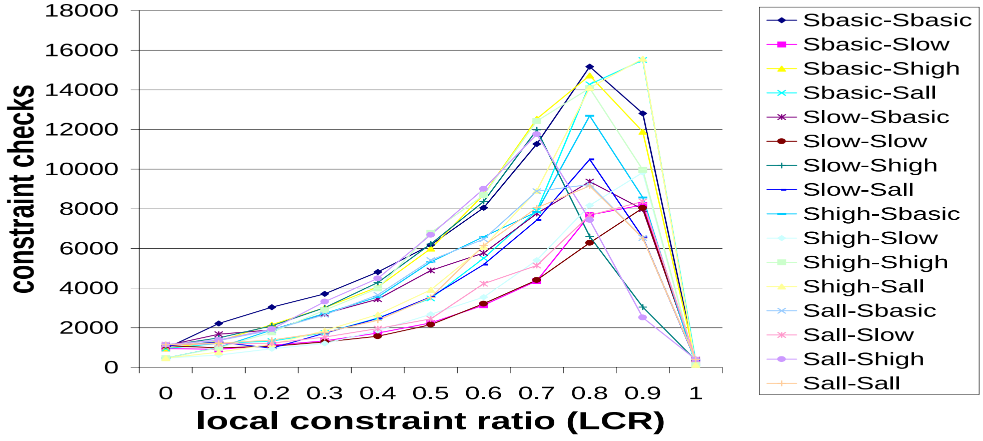


Figure 15: Comparing conflict resolution strategies: constraint checks

between two agents, they essentially formed a chain. In a grid configuration, each agent is externally constrained with four neighbors except for the ones on the grid boundary. All agents share an identical binary constraint by which a value in an agent is not compatible with a set of values in its neighboring agent. This grid configuration is motivated by the distributed sensor networks where multiple agents must collaborate to track targets.

Our experiments followed the method used in [22] and same criteria were used for evaluation. In particular, evaluations were performed by measuring *cycles* and *constraint checks*. *Cycles* is the number of *cycles* (each cycle consists of value communication and assignment) consumed until a solution is found, and *constraint checks* (to measure the total computation cost) is the sum of the maximal numbers of constraint checks performed by agents at each cycle. Experiments were performed for the 16 conflict resolution strategies described in Section 6.2. The number of agents was 512 and the domain size of each agent is 36. The experimental results reported below were from 500 test runs and all the problem instances were solvable with multiple solutions

6.4 Performance of conflict resolution strategies

Conflict resolution strategies described in Section 6.2 were compared on both configurations. In Figure 15, *constraint checks* is shown for all the 16 strategies. The horizontal axis plots the ratio of the number of locally constrained agents to the total number of agents. Each locally constrained agent has a local constraint which restricts available values its domain into a randomly selected contiguous region. For example, local constraint ratio 0.1 means that 10 percent of the agents have local constraints. Local constraint ratio will henceforth be abbreviated as LCR. Having local constraints, agents have less choices to select in conflict resolution. The vertical axis plots the number of *constraint checks*. The results for all the 16 strategies on both configurations showed that S$_{low}$-S$_{low}$ or S$_{low}$-S$_{high}$ was the best, and the results also showed that those strategies with S$_{high}$ or S$_{basic}$ for the *good* case performed worse than the others.

Given 16 strategies, it is difficult to understand different patterns in Figure 15. For expository purposes, we will henceforth present the results from four specific strategies. First, S$_{all}$-S$_{all}$ is selected because it is the maximally cooperative strategy. Second, the original AWC strategy (S$_{basic}$-S$_{basic}$) is selected to compare it with other strategies. Third, S$_{low}$-S$_{low}$ is selected because it showed the best overall performance. Lastly, S$_{low}$-S$_{high}$ is selected because it performed better than S$_{low}$-S$_{low}$ in a special case. Using these four strategies does not change the conclusions from our work, rather it is done solely for expository purpose.

Figure 16 shows the *constraint checks* of the selected strategies on both configurations described above. These graphs show the interesting result that maximal cooperativeness towards neighboring agents is not the best strategy. Though S$_{all}$-S$_{all}$ performed better than S$_{basic}$-S$_{basic}$, it was worse than other less cooperative strategies. More specifically, S$_{low}$-S$_{low}$, one of the lower level cooperative strategies from Figure 14, showed the best performance except for the LCR of 0.0 and 0.9. (S$_{all}$-S$_{all}$ and S$_{low}$-S$_{high}$ were better than S$_{low}$-S$_{low}$ at 0.9. (S$_{all}$-S$_{all}$ and S$_{low}$-S$_{high}$ were better than S$_{low}$-S$_{low}$ at 0.0 in the chain and at 0.9 in the grid, respectively.) While S$_{all}$-S$_{all}$ did not win in terms of *constraint checks*, one possible explanation is its overhead in value selection - thus, it may still win in the number of *cycles*. However, the results in Figure 17 eliminate such a possibility because *cycles* show that S$_{all}$-S$_{all}$ was not the winner: even in terms of *cycles*, it was equivalent to or worse than other less cooperative strategies.

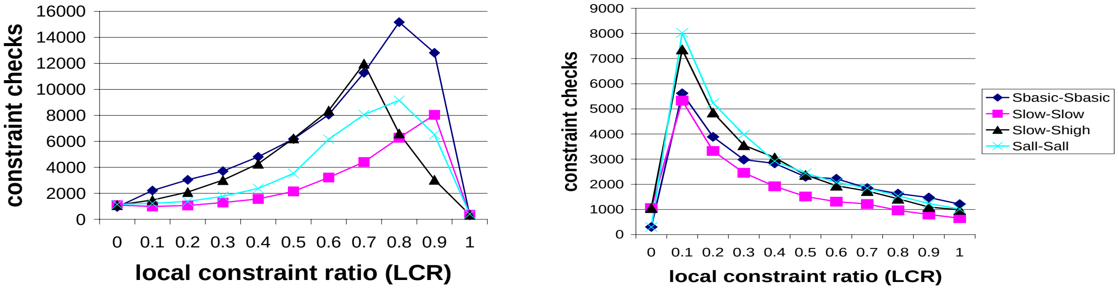
The results above are surprising because, in cooperative environments, we expected that the most cooperative strategy S$_{all}$-S$_{all}$ would perform the best. However, much less cooperative strategy S$_{low}$-S$_{low}$ or S$_{low}$-S$_{high}$ showed the best performance. We conclude that a certain level of cooperativeness is useful, but even in fully cooperative settings, maximal cooperativeness is not necessarily the best strategy.

A related key point to note is that choosing the right strategy has significant impact on convergence. Certainly, choosing S$_{basic}$-S$_{basic}$ may lead to significantly slower convergence rate while appropriately choosing S$_{low}$-S$_{low}$ or S$_{low}$-S$_{high}$ can lead to significant improvements in convergence. For instance, between S$_{basic}$-S$_{basic}$ and the best cooperative strategy in the grid configuration, max average difference was 4-fold in *constraint checks* and 7-fold in *cycles*. For some individual cases, there was more than 30-fold speedup in *constraint checks* and *cycles*.

Here, to check the statistical significance of the performance difference, two-tailed t-test was done with the following two null hypotheses. For the null hypotheses, let $\mu_{\alpha-\beta}$ be an average *constraint checks* (or *cycles*) of a strategy S$_\alpha$-S$_\beta$.

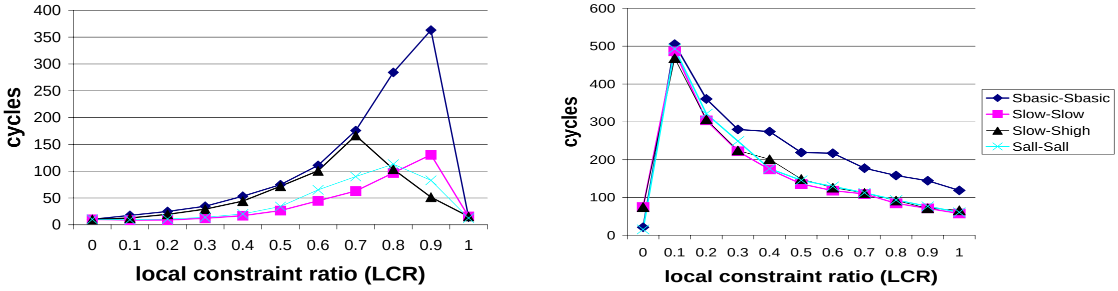
1. $\mu_{basic-basic} = \mu_{low-low}$
2. $\mu_{all-all} = \mu_{low-low}$

The t-test was done at each LCR in the chain and the grid: at the LCR of 0.9 in the grid, $\mu_{low-low}$ was replaced with $\mu_{low-high}$. Both null hypotheses above are rejected, i.e., the differences are significant, with p-value < 0.01 for all values of LCR (one exception is in the chain, p-value at LCR of 0.1 is less than 0.03).



(a) Grid (b) Chain

Figure 16: Comparing conflict resolution strategies: constraint checks



(a) Grid (b) Chain

Figure 17: Comparing conflict resolution strategies: cycle

7 Conclusion

Despite the significant progress and attention paid to distributed resource allocation, it is still an open, challenging problem in multiagent systems research. In this work, we have identified three major shortcomings of current research: a) lack of formalisms and general distributed constraint reasoning (DCR) solution strategies for distributed resource allocation, b) lack of sufficiently advanced DCR algorithms, and c) lack of understanding of principles necessary for applying DCR to real-world hardware. We have presented three contributions for addressing these shortcomings: a) a formalism of distributed resource allocation that allowed a detailed complexity analysis and generalized mappings into DCR, b) an asynchronous algorithm for DCR applicable in overconstrained situations, and c) a two-layered architecture for applying DCR to real-world hardware. Indeed, future researchers attempting to solve difficult distributed resource allocation problems will find assistance in this chapter in understanding the difficulty of their problem, choosing an appropriate DCR representation and algorithm, and successfully applying the algorithm in a real-world application.

8 Technology Transfer

We are looking into ways to transfer our Distributed Constraint Reasoning (DCR) technology into other arenas.

8.1 Multi-Spacecraft Missions for NASA

We are looking at using the DCR framework to represent and reason about fault diagnosis in multi-spacecraft missions. The goal is to use distributed optimization to allow multiple spacecraft to perform distributed error diagnosis. The DCR algorithm allows the spacecraft to resolve inconsistencies between their local sensor data and remote information received from other spacecraft in order to estimate the most probable cause of system errors. With advanced DCR technologies, the multiple spacecraft can recover from errors and continue with science missions with minimal guidance from earth-based human operators.

8.2 Disaster Rescue

We are also looking into the use of the DCR framework for ensuring the coordination of rescue personnel responding to an urban disaster. We again use agents to represent the personnel (e.g., fire trucks, policemen, ambulances) and the DCR algorithms to effectively coordinate the agents in optimally allocating their resources to dynamic tasks such as fires or other disaster sites.

9 Journal, conference, symposia publications

- Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, Makoto Yokoo. Solving Distributed Constraint Optimization Problems Optimally, Efficiently and Asynchronously Submitted to Journal of Artificial Intelligence, February 2003.

- Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe. Distributed Resource Allocation: Formalization, Complexity Results and Mappings to Distributed CSPs Submitted to Journal of Autonomous Agents and Multi-Agent Systems, November 2002.

- Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe. A Dynamic Distributed Constraint Satisfaction Approach to Resource Allocation To appear in *Distributed Sensor Networks: A Multiagent Perspective,* Kluwer Publishers, 2003.

- P.J. Modi, WM Shen, M. Tambe, M. Yokoo An Asynchronous Complete Method for Distributed Constraint Optimization In *Proceedings of Autonomous Agents and Multi-Agent Systems*, Melbourne, Australia, 2003.

- P. Scerri, P.J. Modi, W. Shen, M. Tambe Are multiagent algorithms relevant for robotics applications? A case study of distributed constraint algorithms ACM Symposium on Applied Computing, 2003.

- Pragnesh Jay Modi, Hyuckchul Jung, Milind Tambe, Wei-Min Shen, Shriniwas Kulkarni A Dynamic Distributed Constraint Satisfaction Approach to Resource Allocation Proceedings of Seventh International Conference on Principles and Practice of Constraint Programming, Paphos, Cyprus, 2001.

- Hyuckchul Jung, Milind Tambe, Shriniwas Kulkarni Argumentation as Distributed Constraint Satisfaction: Applications and Results, Proceedings of the International Conference on Autonomous Agents, 2001.

- Pragnesh Jay Modi, Hyuckchul Jung, Milind Tambe, Milind Tambe, Wei-Min Shen, Shriniwas Kulkarni, Dynamic Distributed Resource Allocation: A Distributed Constraint Satisfaction Approach Intelligent Agents VIII: Agent Theories, Architectures and Languages (ATAL), Springer Lecture Note in Computer Science, Volume 2333, 2001.

10 Personnel

- Milind Tambe (PI)
- Wei-min Shen (Co-PI)
- Jay Modi (GRA)
- Hyuckchul Jung (GRA)
- Paul Scerri (Post-doc)

References

[1] R. Caulder, J.E. Smith, A.J. Courtemanche, M.F. Mar, and A.Z. Ceranowicz. Modsaf behavior simulation and control. In *Proceedings of the Conference on Computer Generated Forces and Behavioral Representation*, 1993.

[2] S.E Conry, K. Kuwabara, V.A. Lesser, and R.A. Meyer. Multistage negotiation for distributed constraint satisfaction. *IEEE Trans. Systems, Man and Cybernetics*, 1991.

[3] K. Decker and J. Li. Coordinated hospital patient scheduling. In *Proceedings of International Conference on Multi-Agent Systems*, 1998.

[4] Y. Hamadi, C. Bessiere, and J. Quinqueton. Backtracking in distributed constraint networks. In *European Conference on Artificial Intelligence*, 1998.

[5] Steve Hanks, Martha Pollack, and Paul Cohen. Benchmarks, testbeds, controlled experimentation, and the design of a gent architectures. *AI Magazine*, 14(4):17–42, 1993.

[6] K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In G. Smolka, editor, *Principles and Practice of Constraint Programming*, pages 222–236. 1997.

[7] K. Hirayama and M. Yokoo. An approach to over-constrained distributed constraint satisfaction problems: Distributed hierarchical constraint satisfaction. In *Proceedings of International Conference on Multiagent Systems*, 2000.

[8] H. Jung, M. Tambe, and S. Kulkarni. Argumentation as distributed constraint satisfaction: Applications and results. In *Proceedings of the International Conference on Autonomous Agents*, 2001.

[9] H. Kitano, S. Todokoro, I. Noda, H. Matsubara, and T Takahashi. Robocup rescue: Search and rescue in large-scale disaster as a domain for autonomous agents research. In *Proceedings of the IEEE International Conference on System, Man, and Cybernetics*, 1999.

[10] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[11] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[12] P. Mesequer and M. A. Jiménez. Distributed forward checking. In *Proceedings of CP-00 Workshop on Distributed Constraint Satisfaction*, 2000.

[13] S. Minton, M. D. Johnston, A. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the National Conference on Artificial Intelligence*, 1990.

[14] P. J. Modi, H. Jung, W. Shen, M. Tambe, and S. Kulkarni. A dynamic distributed constraint satisfaction approach to resource allocation. In *Principles and Practice of Constraint Programming*, 2001.

[15] P. J. Modi, W. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for general distributed constraint optimization. In *Proc of Autonomous Agents and Multi-Agent Systems Workshop on Distributed Constraint Reasoning*, 2002.

[16] V. Parunak, A. Ward, M. Fleischer, J. Sauter, and T. Chang. Distributed component-centered design as agent-based distributed constraint optimization. In *Proc. of the AAAI Workshop on Constraints and Agents*, 1997.

[17] P. Scerri, P.J. Modi, W. Shen, and M. Tambe. Are multiagent algorithms relevant for robotics applications? a case study of distributed constraint algorithms. In *ACM Symposium on Applied Computing*, 2003.

[18] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000).*

[19] Katia Sycara, Steven F Roth, Norman Sadeh-Koniecpol, and Mark S. Fox. Distributed constrained heuristic search. *IEEE Transactions on Systems, Man, and Cybernetics*, 21:1446–1461, 1991.

[20] Michael Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, pages 1–23, 1993.

[21] M. Yokoo. *Distributed Constraint Satisfaction:Foundation of Cooperation in Multi-agent Systems*. Springer, 2001.

[22] M. Yokoo and K. Hirayama. Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of International Conference on Multiagent Systems*, 1998.

[23] W. Zhang and L. Wittenburg. Distributed breakout revisited. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2002.